The Wang Professional Computer FORTRAN Reference Manual

First Edition — July 1983

700-8316

WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL: 617/459-5000, TWX 710-343-6769, TELEX 94-7421

Disclaimer of Warranties and Limitation of Liabilities

The staff of Wang Laboratories, Inc., has taken due care in preparing this manual; however, nothing contained herein modifies or alters in any way the standard terms and conditions of the Wang purchase, lease, or license agreement by which this software package was acquired, nor increases in any way Wang's liability to the customer. In no event shall Wang Laboratories, Inc., or its subsidiaries be liable for incidental or consequential damages in connection with or arising from the use of the software package, the accompanying manual, or any related materials.

NOTICE:

All Wang Program Products are licensed to customers in accordance with the terms and conditions of the Wang Laboratories, Inc. Standard Program Products License; no ownership of Wang Software is transferred and any use beyond the terms of the aforesaid License, without the written authorization of Wang Laboratories, Inc., is prohibited.

WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL: 617/459-5000, TWX 710-343-6769, TELEX 94-7421

PREFACE

This is a reference manual for Wang Professional Computer (PC) FORTRAN. Wang PC FORTRAN conforms to the standard ANSI X3.0-1978 at the subset level. Wang PC FORTRAN includes extensions to the standard language as well as some of the features of the full ANSI standard.

This manual presents the rules and syntax of Wang PC FORTRAN. It does not explain how to program in FORTRAN. To use this manual, you should possess a previous understanding of FORTRAN. Any basic FORTRAN text book can provide you with this knowledge.

Chapter 1 describes how to run a FORTRAN program on the Wang Professional Computer. It explains how to invoke the two passes of the FORTRAN compiler, how to name your program, and how to invoke the Linker. It also describes the format of the source listing file and explains how to read it.

Chapter 2 is general in scope. It details the elements of the source file, including the character set, lines, columns, program units, data types, and FORTRAN names.

Chapter 3 defines expressions: arithmetic, character, and logical. Chapter 4 describes DATA and ASSIGNMENT statements. Chapters 5 and 6 outline programming statements. Chapter 5 deals with specification statements, while Chapter 6 deals with control statements. Chapters 7 and 8 describe input and output requirements. Chapter 7 discusses records, files, and input and output statements. The format statement, data, repeat, literal, and spacing descriptors are discussed in Chapter 8.

Chapter 9 describes programs, subroutines, and functions and their associated statements. Chapter 10 deals with the metacommands that are part of Wang PC FORTRAN.

There are four appendixes. Appendix A lists error messages; Appendix B describes how Wang PC FORTRAN conforms to the full language features of FORTRAN 77 and to the extensions of the standard; Appendix C tells how to interface Wang PC FORTRAN with assembly language; and Appendix D presents the Wang PC File Control Block 1.

Syntax Notation

In this manual, a general format line accompanies statements, comma directives, and functions. This format uses specific conventions. The conventions are as follows:

- Square brackets ([ ]) indicate that the enclosed entry is optional.
- You must type in all capitalized items as shown.
- The noncapitalized items describe the input that is required. You must supply an entry defined by the text. For example, if the word is expression, you must supply an expression. The types of legal expressions for that command are outlined in the text if the expression differs from the usual definition.
- An ellipsis (...) following an item indicates that you can repeat the item.
- An underlined word after a command indicates user input, i.e., the user typed in the underlined words.

Aside from the previous formatting notations, you must use all punctuation marks such as commas, equal signs, slash marks, quotation marks, or parentheses exactly as shown.

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

TABLES

FIGURES

1

How to Compile, Link and Run a FORTRAN Program

CHAPTER 1
HOW TO COMPILE, LINK, AND RUN A FORTRAN PROGRAM

1.1 INTRODUCTION

The Wang Professional Computer (PC) Compiler has a three-part structure. The first two parts, Pass One and Pass Two, create object code. Pass Three is an optional step that creates an object code listing.

Before you can run your program you must Link it. To compile, link, and run a FORTRAN program on the Wang Professional Computer, you must perform the following steps. After each of the commands, press RETURN to enter the them.

1. Create your source file with the extension .FOR by using the Wang PC Editor.

2. Call Pass One of the compiler and answer the prompts

```
B:FOR1

    Source filename [.FOR]: TEST
    Object Filename [TEST.OBJ]:TEST
    Source listing [NUL.LST]:TEST
    Object listing [NUL.COD]: TEST

            Pass One No Errors Detected
            8 Source Lines
```

3. Call on Pass Two of the compiler and wait for the message

```
B:PAS2

    Code Area Size = #009F   (   159)
    Cons Area Size = #000C   (    12)
    Data Area Size = #002C   (    14)

    Pass Two  No Errors Detected
```

4. Call on Pass Three of the Compiler

```
B:PAS3

   [Object listing written as TEST.COD]
```

5. Call the Linker and respond to the prompts. (In the following example, TEST is linked with two assembly language modules, ASM1 and ASM2.)

```
B: LINK TEST

   Object modules [TEST.OBJ]: A:TEST A:ASM1 A:ASM2
   Run file [TEST.EXE]: TEST
   List file [NUL.MAP]:TEST
   Libraries [.LIB]:FORTRAN
```

6. Run the program by entering the following command and then pressing RETURN.

```
B:TEST
```

1.2 FILE AND DEFAULT EXTENSION NAMES OF PASSES ONE AND TWO

Passes One and Two deal with four separate files. They are as follows.

Table 1-1. The Files of Passes One and Two

File Name	Function	Default Extension
The Source File	The user-created FORTRAN program.	.FOR
The Object File	The translation of the FORTRAN source program into relocatable binary machine code.	.OBJ
The Source Listing File	The source program listing with the names and types of the symbols, the program parts, and the errors present.	.LST
The Object Listing File	A symbolic listing of the object code and its relative addresses. It is read by Pass Three and used for debugging with the DEBUG utility.	.COD

In addition to the permanent files created during Pass One, the compiler creates two intermediate files that it uses during Pass Two:

- PASIBF.SYM
- PASIBF.BIN

If the compiler finds errors during Pass One, it deletes these two files. To continue compilation, you must return to editing mode and correct the errors. If the compiler finds no errors or generates warnings only, it retains these two files. You are now ready to continue with Pass Two.

1.2.1 Pass One

Once the system prompt appears, you can invoke Pass One of the compiler. Enter the command FOR1. This causes the processor to load Pass One into memory. After you enter FOR1, you can enter all the file names on the same line and then press the RETURN key (Method One), or you can press the RETURN key immediately, wait for a series of prompts to appear, and respond to them one by one (Method Two).

Method One

If you choose to write the file names on the same line and then press the RETURN key, the syntax is as follows:

```
FOR1 source file, object file, source listing file, object listing file
```

Upon this command, the compiler compiles your source program, creating an object, listing, and object listing file.

In the following example, the compiler compiles your source file, creating an object file, a source listing file, and an object listing file all named PROG1:

```
B:FOR1 PROG1, PROG1, PROG1, PROG1
```

Method Two

A second way of invoking Pass One of the FORTRAN compiler is to write FOR1 after the system prompt appears.

```
B:FOR1
```

The processor loads Pass One into memory and returns four prompts, one at a time. You answer the prompts with the names of the files that you wish to create. If you are creating a file called PROG1, the following prompts appear. As soon as you answer one, the next one appears.

```
Source filename [.FOR]:
Object filename [PROG1.OBJ]:
Source listing [NUL.LST]:
Object listing [NUL.COD]:
```

Each prompt requires a certain response. The prompts and possible responses are as follows:

Prompt 1 Source filename [.FOR]:

You can respond to this prompt by entering

- The file name, causing your file to use the default extension .FOR.
- The file name and an extension, if the extension is different from the default.

Prompt 2 Object filename [PROG1.OBJ]:

You can respond to this prompt by performing any of the following:

- Pressing RETURN, causing your file to have the same file name as your source program with the default extension .OBJ.
- Entering the file name, so that your file has the same name as the file name that you have entered with the default extension .OBJ.
- Entering the file name and an extension, so that your object file has the same file name and extension as you have just entered.
- Entering another drive, so that the compiler uses the default name and extension, but overrides the default drive.

Prompt 3 Source listing [NUL.LST]:

You can respond to this prompt by any of the following:

- Pressing RETURN, so that you get no listing file;
- Entering a file name, so that the listing file has the same name as the source file with the default extension .LST;
- Entering a file name and an extension name, which creates a listing file with a name extension of that name; or
- Entering another drive, so that the compiler uses the default name and extension, but overrides the default drive.
- Entering USER, so that file is written on the screen immediately as the listing is created.
- Entering CON, so that the listing file appears on the screen in blocks of 512 bytes.

Prompt 4 Object listing [NUL.COD]:

You can respond to this prompt by any of the following:

- Pressing RETURN, so that no file is created;
- Entering a file name and creating a file of that name with the default extension .COD;
- Entering a file name and an extension name, so that the assembler creates a file of that name and extension; or
- Entering another drive, so that the compiler uses the non-null name and extension as the name and extension of your file and the drive you entered rather than the default.
- Entering USER, so that the file appears on the screen as it is created.
- Entering CON, so that the file appears on the screen in 512 byte blocks.

The following is a typical format for responding to the prompts of Method Two:

```
Source filename [.FOR]:  TEST
Object filename [TEST.OBJ]:  A:
Source listing [NUL.LST]: TEST
Object listing [NUL.COD]: RETURN
```

When you use the previous commands, the compiler

- Names the source file TEST.FOR.
- Names the object file, TEST.OBJ and places it on Drive A rather than on the default B.
- Names the listing file TEST.LST, which overrides the default of creating no file at all (a null file).
- Creates no object listing file. In this case, you cannot run Pass Three.

1.2.2 Control Characters

You can replace file names with three other responses: a carriage return, a comma, or a semicolon. The effect of each response is as follows.

• Carriage return	causes the assembler to accept the default name for that file only.
• Semicolon	causes the assembler to accept the default file names for all of the remaining files.
• Comma	causes the assembler to accept the default for that particular file in Method One only (in which you list all four file names after the MASM command).

The Carriage Return

The carriage return accepts the default name for that file only. All of the remaining prompts appear, but the file name and the default extension in the brackets becomes the name of your file. In the following example, PROG1.OBJ becomes the name of your file when you press RETURN.

```
Object filename [PROG1.OBJ]: (RETURN)
```

The Semicolon

When you enter a single semicolon followed by carriage return any time after you have entered the source file prompt, you select the default responses for all of the remaining prompts. After using the semicolon, there are no more prompts. Therefore, if you want to skip over one prompt only, use the carriage return.

Once you enter the semicolon to respond to the object file prompt, as shown in the following example, no further prompts appear. The compiler compiles the source and object files. The defaults for the source listing and object listing files are NUL files. Therefore, the compiler creates no source listing or object listing files.

```
Source filename [.FOR]:  PROG1
Object filename [PROG1.OBJ] ; (RETURN)
```

The Comma

You use the commas with Method One only. The comma causes the compiler to accept the default name for that file only. In the following example, the compiler creates PROG1.FOR, PROG1.OBJ, PROG1.LST, and PROG1.COD.

```
FOR1 PROG1.FOR,,PROG1, PROG1
```

There are two other controls that help you in the creation of your FORTRAN program. Each causes you to exit from the mode that you are in and return to the system prompt. They are as follows:

CONTROL + C	Returns you to command level. If you are in the assembly program, CONTROL and C causes compiling to stop. Once back at command level, you can invoke the compiler again.
SHIFT + CANCEL	Returns you to command level.

1.2.3 Invoking Pass One

After you enter the last file name by either method, Pass One begins to compile your program. When it is finished, a list of warning and severe errors appears. The following is an example of an error message:

```
*****  Error 163, line 8--FORMAT NOT FOUND
          Pass One      1 Errors Detected
                        8 Source Lines
```

You must return to editing mode to correct the errors.

If the following message appears, Pass One has compiled your program successfully and you are ready for Pass Two

```
Pass One  No Errors Detected
```

1.2.4 Invoking Pass Two

Pass Two completes compilation of your program by

- Reading PASIBF.SYM and PASIBF.BIN
- Writing and reading the temporary file PASIBF.TMP
- Writing two new intermediate files for use in Pass Three
- Deleting the intermediate and temporary files of Pass One

PAS2.EXE assumes that the intermediate files created in Pass One are on the default drive. If you have switched disks, you must indicate their location on the command that starts Pass Two. For example:

```
A:PAS2A/PAUSE
```

The "A" tells the compiler that the intermediate files are on drive A, and the "/PAUSE" tells the compiler to pause so that you can insert the disk that contains the files into Drive A. After pausing, Pass Two prompts as follows:

```
Press enter key to begin pass two
```

After you insert the new disk in drive A, press RETURN and the compiler proceeds with Pass Two. To invoke Pass Two, enter the following command after the system prompt appears:

```
B:PAS2
```

Pass Two does not prompt you for input. When it is completed, the following message appears:

```
Code Area Size = #hhhh     (dddd)
Cons Area Size = #hhhh     (dddd)
Data Area Size = #hhhh     (dddd)

  Pass Two       No Errors Detected
```

Pass Two sends out messages indicating the amount of space taken up by executable code (Code), constants (Cons), and variables (Data) in both hexadecimal (#hhhh) and decimal (dddd) form. The sum of the Code area size and the Cons area size gives the number of bytes of ROMable code. The data area size gives the number of bytes of non-ROMable memory.

The error message applies only to Pass Two. If you have errors, you must correct them in editing mode. If you have no errors, you are ready for linking.

1.2.5 Invoking Pass Three

Pass Three reads the two temporary files that the compiler created when you entered a file name at the last prompt (object listing) of Pass One. If you did not create these files, you cannot run Pass Three. Be sure the disk with Pass Three is in the current drive; then start Pass Three by typing

```
A:PAS3
```

PAS3.EXE does not prompt you for any input. It reads the two temporary files, PASIBF.TMP and PASIBF.OID, and writes the object code listing to the file DEMO.COD.

When Pass Three is complete, the two temporary files are deleted. If you decide not to run Pass Three after requesting an object listing, you should delete these files yourself to save space.

1.2.6 Linking Your Program

The Linker links subroutines, library routines, and assembly language routines to your main program module. To use the Linker, put the utility diskette that includes the Linker and the runtime library into Drive B. If PAS2.EXE is on this diskette, this step is not necessary. Enter

```
B:LINK TEST
```

The Linker responds with a series of prompts as listed in Table 1-2.

Table 1-2. Prompts for Pass Three

Prompt	Possible Responses
Object modules [TEST.OBJ]:	FORTRAN file name first followed by Pascal and then assembly language modules.
Run file [TEST.EXE]:	TEST creates TEST.EXE. RETURN accepts the default TEST.EXE. ; accepts the defaults for the rest of the prompts.
List file [NUL.MAP]:	TEST creates TEST.MAP. CON allows you to see this file on your console, but it is not written on the disk. RETURN creates the default of no file. ; accepts the rest of the defaults.
Libraries [.LIB]:	B:; indicates your library is on Drive B rather than the default drive. FORTRAN.LIB searches FORTRAN.LIB if it is on the default drive. RETURN searches FORTRAN.LIB if it is on the default drive. PASCAL.LIB searches the Pascal library. PASCAL.LIB+BASIC.LIB searches the Pascal and the BASIC libraries.

The primary output of the linking process is an executable run file. ou can request a Linker map or a listing file as well. The Linker map shows e addresses, relative to the start of the run module, for every code or data gment in your program. It also includes all EXTERN and Public variables. request the map, use the map switch /MAP.

If you press RETURN in response to the Library [.LIB] prompts, the Linker automatically searches for a library called FORTRAN.LIB on the default drive. If FORTRAN.LIB is not there, the following message appears:

```
Cannot find library FORTRAN.LIB
Enter new drive letter:
```

Find the disk with FORTRAN.LIB, and enter it into the default drive. If you press the RETURN key when the preceding message appears, Linking will proceed without a library search.

If you instruct the Linker to search other libraries, enter the library names separated by plus signs in response to the library prompts.

After you have responded to the last of the four prompts, the Linker links your compiled program with the necessary modules in the FORTRAN runtime library. This linking process creates an executable file on the default drive. To assure correct linkage and to save space in memory, you should perform the following steps:

- Set the /DSALLOCATION switch (this is the default).
- Set the /LOW switch.
- Load the FORTRAN object file first; if you load an assembly language module first, the Linker may order the segments incorrectly.
- If you have no READ or WRITE statements, and if you use no file system routines, declare dummy procedures to eliminate procedures that initialize and close the file system. The dummy subroutines are as follows:

Table 1-3. Dummy Subroutines in the Linking Process

Dummy Subroutine	Function of Real Subroutine	Function Dummy Performs
SUBROUTINE INIVQQ RETURN	Initializes the file system.	Saves the space that would be necessary if the Linker brought in the real subroutine that initializes data.
SUBROUTINE ENDYQQ RETURN	Closes all files when a program terminates.	Saves the space that would be necessary if the linker brought in the real subroutine that closes all files when a program terminates.

PC FORTRAN programs that perform I/O require linking to the PC FORTRAN ile system in the runtime library. INIVQQ initializes the file system, and NDYQQ terminates the file system. If your program does not use the file ystem, it does not need to use INIVQQ or ENDYQQ. The Linker uses these two ubroutines automatically, however. You can eliminate the increased size aused by linking and loading routines that you do not need to use by eclaring dummy INIVQQ and ENDYQQ subroutines. The dummy subroutine will not ork if the Linker produces either of the following messages:

- Any "Undefined Global" messages for global names that end with the VQQ or UQQ suffix
- A "Multiply Defined Global" message for INIVQQ

inker Switches

ou can use two switches with the Linker. Their names and functions are isted in Table 1-4.

Table 1-4. Linker Switches

Switch	Function
/DSALLOCATE	Directs the Linker to load all data at the high end of the data segment. The presence of the/DSALLOCATE switch coupled with the absence of the /HIGH allows your program to allocate dynamically. This switch is required for PC FORTRAN.
/LINENUMBERS	Causes the Linker to include all source line numbers and their associated addresses in its listing.
/MAP	Directs the Linker to list all public symbols defined in input modules together with their definitions.
/NO	Short for NODEFAULTLIBRARYSEARCH. Prevents the Linker from searching the Default Library.
/PAUSE	Causes the Linker to pause in the link session until you press RETURN.
/PUBLICS	Causes the Linker to include all EXTERN and PUBLIC variables in its listing
/STACK:number	Directs the Linker to allocate a stack for the run file of the size specified (number) up to 65,536 bytes.

When the Linker is finished, it sends out a series of error messages. o run your program, return to editing mode, correct the errors, and eassemble and relink your program.

For more information concerning the Linker and its utilities, see The Wang Professional Computer Program Development Guide, 700-8018.

1.2.7 Running the Program

To run the executable program, return to the system prompt and enter the name of the program. The extension name is not required.

```
B: PROG
```

This command directs the operating system to load the executable file, fix segment addresses to their absolute value, and start execution.

1.3 THE SOURCE LISTING FILE

The source listing file provides valuable information for debugging your program. There are two distinct parts to the source listing file: the listing of the program and the listing of the symbols.

1.3.1 The Program Listing

The program listing of the source listing file includes

- the source program
- the line number of each line of the source program
- runtime errors if you have used the the $DEBUG metacommand (Refer to Section 10.1 for more information on the $DEBUG metacommand.)
- the column number of columns 1 and 7 and an indication of column 72

The following is an example of a program listing:

```
D  Line#  1     7
       1        INTEGER N,J
       2        N=1
       3  15    J=N*N
       4        WRITE (*,20) N,J
       5  20    FORMAT ('The SQUARE ROOT OF' I3 '=' I3)
       6        N=N+1
       7        IF (N.LE.10) GOTO 15
       8        STOP
       9        END
```

Format

You can control specific parts of the format such as the listing of the title and the subtitle, the source line that the compiler lists or does not list, and the size of the line and the page through the use of metacommands. For specific information about the formatting metacommands, see Chapter 10, Metacommands.

The Column Numbers

Column numbers 1 and 7 appear on the line directly above the source program. The compiler truncates lines that extend beyond 72 columns to indicate that they are too long.

The D Column Label

The D column contains the current nesting level of DO loops. It can help you find missing terminal statements in DO loop constructions. The compiler increments the number in the D column for the statement following the DO statement and decrements it following the terminal statement. If two or more DO loops share the same terminal statement, the compiler places the innermost DO loop on that terminal statement. In the following listing, the D column is empty because the DO nest level is zero.

1.3.2 The Symbolic Listing

The symbol table of the source program listing has two parts: the local variable listing table and the global listing table.

The local variable symbol table lists five categories: name, type, offset, P (parameter), and class. The global symbol table lists four categories: name, type, size, and class. The following is the local and global symbol table for the program above:

Table 1-5. The Symbol Table

Name	Type	Offset	P	Class
J	INTEGER*4		6	
N	INTEGER*4		2	
Name	Type		Size	Class
MAIN				PROGRAM

To the far left, the compiler lists the names of the variables or program units in alphabetical order. Next to the name is the data type, which is reserved for variables and functions. The data types are the same types discussed in Chapter 2.

The offset is the offset address in bytes from the beginning of the segment or block in which the variable is defined. There are four possible program segments: data, code, stack, and extra. In addition, an element within a common block has an offset within the common block. Table 1-6 shows which segment the offset represents with each data type.

Table 1-6. The Offset of Data Types

Data Type	Source of Offset
variable	data or extra segment
common block	common block
formal parameter	stack

If the offset entry is a set of asterisks, you have defined the variable without referencing it within the program unit.

The parameter (P) can be either a blank or an asterisk. The presence of an asterisk indicates that the name is a formal parameter.

Class indicates whether the name is a function, a subroutine, common, intrinsic, external, a parameter, a common block, a variable, or a variable within a common block.

Size is reserved for global symbols and denotes the size of the common block.

1.4 USING A BATCH COMMAND FILE

The batch file facility lets you create a batch file for executing a series of commands. A batch command file is a text file of operating system commands. If a batch file is open when the operating system is ready to process a command, the next line in the file becomes the command line. After processing all batch command lines (or if batch processing is otherwise terminated), the operating system goes back to reading command lines from the screen.

Batch file lines cannot be read by the compiler, the linker, or a user program. Thus, you cannot put prompts into a batch file. All compiler arguments must be given on the command line. The batch file can contain dummy arguments that you replace with actual arguments when you invoke it. The symbol %1 refers to the first parameter on the line, %2 to the second parameter, and so on. The limit is %9. A batch command file must have the extension .BAT and should be kept on either the program disk or the utility disk.

The PAUSE command, followed by the text of the prompt, tells the operating system to pause, display a prompt (which you have defined), and wait for some further input before continuing. If your program is already debugged and you are making only minor changes to it, you can speed up the compilation process by creating a batch file that issues the compile, link, and run commands. For example, you can create the following batch file, COLIGO.BAT:

```
A:FOR1 %1,,;
PAUSE ...If no errors, insert PAS2 disk in drive A:.
A:PAS2
PAUSE ...Insert runtime libraries disk in drive A:.
A:LINK %1;
%1
```

The previous file operates as follows.

1. The first line of the batch file runs Pass One of the compiler.

2. The second line generates a pause and prompts you to insert the pass two disk.

3. The third line runs Pass Two.

4. The fourth line generates a pause and prompts you to insert the runtime library.

5. The fifth line links the object file.

6. The sixth line runs the executable file.

A BAT file is only executed if there is neither a COM file or EXE file with the same name. Thus, if you keep your source file and BAT file on the same disk, give them different filenames. To execute the file above, enter

```
COLIGO DEMO
```

DEMO is the name of the source program you want to compile, link, and run.

2

The Source Program

CHAPTER 2
THE SOURCE PROGRAM

2.1 INTRODUCTION

A FORTRAN source program is composed of a sequence of characters that the compiler interprets as identifiers, labels, constants, lines, and statements. This chapter defines the components of a FORTRAN source program.

2.2 CHARACTER SET

A FORTRAN source program is a sequence of coded letters, numbers, and special characters written in ASCII code. The ASCII character set includes the following:

- 52 upper- and lowercase letters (A through Z and a through z)
- 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9
- special characters such as carriage returns, brackets, and punctuation marks -- the remaining printable characters of the ASCII character set

The FORTRAN compiler equates uppercase and lowercase letters except in literal strings of text such as character constants and in Hollerith fields. For an example, the FORTRAN compiler reads all of the following characters strings to be the same:

ABCDE abcde AbCdE aBcDe

The collating sequence for the FORTRAN character set is the ASCII sequence. In the ASCII sequence, the numbers 0-9 are 48-57; the uppercase letter A-Z are 65-90; and the lowercase letters a-z are 97-122. Together, the letters and numbers are called the alphanumeric characters. For further details, refer to Appendix J, ASCII Character Codes.

2.3 FORTRAN NAMES

The compiler recognizes arrays, variables, functions, and subroutines by FORTRAN names. A FORTRAN name, or identifier, consists of a sequence of up to 660 alphanumeric characters (66 per line times 10 lines). The initial character must be alphabetic; subsequent characters must be alphanumeric. Blanks are skipped. Only the first six characters are significant, the rest are ignored. There are no reserved names in FORTRAN. The compiler recognizes keywords by their context. Thus, a program can have an array named IF, READ, or GOTO as long as the name conforms to the rules.

2.3.1 Scope of FORTRAN Names

FORTRAN names can be global, and you can reference them from outside the particular program part in which they appear, or they can be local, and you can only reference them from within the program part.

All subroutine, function subprogram, common block, and program names have global scope. Therefore, you cannot use the same name for them more than once in the same program.

A name with local scope is only known within a single program unit. Therefore, you can repeat a name with local scope in other program parts with no conflict. The names of variables, arrays, parameters, and statement functions all have local scope. There are two exceptions to the above rules:

- Common block names are always enclosed in slashes, such as /NAME/. The compiler, therefore, can tell the difference between the common block name and a local name even if both are the same.

- The scope of statement function parameters is limited to the single statement forming that statement function. You cannot repeat the same name within the statement function itself, but you can use the same name outside of it.

2.3.2 Undeclared FORTRAN Names

The compiler classifies names by their syntax and the context in which they appear. If you use the name as a variable, the compiler creates an entry in the symbol table for a variable of that name, inferring its type from the first letter of its name. Normally, variables beginning with the letters I, J, K, L, M, or N are integers, while all others are real. You can override these defaults by an IMPLICIT statement. (See Section 5.1 for more information on the IMPLICIT statement.)

If an undeclared name is used as a function call, the compiler creates a symbol table entry for a function of that name. Its type is inferred in the same manner as the type of a variable.

Similarly, a subroutine entry is created for a newly encountered name that is the target of a CALL statement. If an entry for such a subroutine or function name exists in the global symbol table, its attributes are coordinated with those of the newly created symbol table entry. If any inconsistencies are detected, such as a previously defined subroutine name being used as a function name, an error message is issued.

If you use an undeclared name as a function call, the compiler creates a symbol table entry for a function of that name. The compiler infers its type by the context in which the name appears, as well as by its syntax.

Similarly, when you create a subroutine entry in a CALL statement, the compiler checks the global symbol table. If the name exists there, the compiler coordinates the attributes of the symbol with those of the newly created symbol table entry. If it detects any inconsistencies, such as a previously defined subroutine name being used as a function name, the compiler issues an error message.

2.4 LINES

A FORTRAN source program is a sequence of lines, each ending with a RETURN. The compiler treats only the first 72 characters in a line as significant, ignoring any trailing characters, and padding the shorter lines with blanks to 72 characters.

2.5 COLUMNS

The column in which a character resides is significant in FORTRAN. Column 1 is for comment and metacommand indicators, columns 1 through 5 for statement labels, column 6 is for continuation indicators, and columns 7 through 72 are for statements and commands. Table 2-1 illustrates the use of columns in the command line in FORTRAN.

Table 2-1. Field Placement

Fields	Columns	Character
label	1-5	an integer no more than 5 digits long
comment	1-5	C, *
continuation statement	6	any character
initial line	6	zero or blank
comment	1-72	all blank
statement	7-72	any FORTRAN statement or command

2.6 BLANKS

Usually, you can use blanks to improve readability in FORTRAN. However, you cannot use blanks in the following instances without affecting the program.

- Within string constants
- Within Hollerith fields
- In column six (since a blank here distinguishes initial lines from continuation lines)

2.7 FORTRAN LINES

Labels, statements, and comments compose FORTRAN lines. These elements can appear in initial or continuation lines. Each element of a FORTRAN statement appears in a certain field according to the rules of FORTRAN. The FORTRAN compiler requires you to use certain rules of syntax as you compose each line and statement. These rules are stated in this section.

2.7.1 Labels

Labels appear in columns one through five of an initial line and are between one and five positive integers long. You use labels to reference statements from other statements. The rules for labels are as follows:

- The label can be any integer from 1 to 99999.
- Leading zeros and blanks are not significant.
- You can use the same label only once within the same program unit.
- The processor recognizes labels only on initial lines.

2.7.2 Initial Lines

The initial line is the first (or only) line of each statement. With the exception of the statement following a logical IF, FORTRAN statements begin with initial lines. An initial line

- Is any statement line other than a comment or a metacommand.
- Contains a blank or a zero character in column six.
- Can be blank or contain a label in the first five columns.
- Can begin anywhere within the statement field.

2.7.3 Continuation Lines

You can use up to 19 continuation lines when you need more space to complete the coding that you began in the initial line. In a continuation line, the first five columns are blank. The sixth column must contain a character other than a blank or a zero.

2.7.4 Comment Lines

Comment lines explain programs or make them more readable without affecting execution. The compiler recognizes a comment line when

- A "C" (or "c") appears in column one
- An "*" appears in column one
- The line contains all blanks

You must follow comment lines with an initial line or another comment line, but not by a continuation line. Extra blank lines at the end of a FORTRAN program result in a compiletime error. The system interprets the blanks as comment lines that are not followed by initial lines.

2.7.5 Tabs

The tab character has the following significance in the source program:

- When the TAB is columns 1 through 5, the next character is considered to be column 7.

- Except when it appears in a character constant or in a Hollerith field, a TAB in columns 7 through 72 is considered to be a blank.

- A TAB in a character constant or a Hollerith field is interpreted as an ASCII TAB character to allow the program to output tabs.

2.7.6 Statements

A FORTRAN statement consists of an initial line, followed by zero to nine continuation lines. A statement can contain as many as 660 characters in columns 7 through 72. The END statement must occupy an initial line with no following continuation lines.

There are two types of FORTRAN statements: executable and nonexecutable. Executable statements cause the compiler to generate object program instructions. The three types of executable statements are

- Assignment statements
- Control statements
- Input/Output statements

Nonexecutable statements provide information to the compiler about data initialization, data types, and input and output formats. The five types of nonexecutable statements are

- Specification statements
- Data initialization statements
- FORMAT statements
- FUNCTION statements
- Subprogram statements

Table 2-2 illustrates the use of the different types of statements in PC FORTRAN.

Table 2-2. Categories of Statements in FORTRAN

Category	Description
Assignment	Executable. Assigns a value to a variable or an array element.
Comment	Nonexecutable. Allows comments within program code.
Control	Executable. Controls the order of execution of statements.
DATA	Nonexecutable. Assigns initial values to variables.
FORMAT	Nonexecutable. Provides data editing information.
I/O	Executable. Specifies sources and destinations of data transfer and other facets of I/O operation.
Specification	Nonexecutable. Defines the attributes of variables, arrays, and function names.
Statement Function	Nonexecutable. Defines a simple, locally used function.
Program Unit Heading	Nonexecutable. Defines the start of a program unit and specifies its formal arguments.

2.8 PROGRAM UNITS

A program unit is a self-contained portion of a program. There are three types of program units: main programs, function subprograms, and subprograms. Each FORTRAN program is composed of a main program or a main program with one or more subprograms.

2.8.1 Main Program and Subprogram

A subprogram is a program unit that you call upon from another program unit to perform some specific, limited function. The subprogram performs the function and then switches control back to the main routine (or sometimes to another subprogram).

A subprogram begins with a SUBROUTINE or a FUNCTION statement and ends with an END statement. A main program begins with a PROGRAM statement or any statement other than a SUBROUTINE or FUNCTION statement and ends with an END statement. The subprogram and the main program are called program units.

2.8.2 Statement Ordering

The compiler recognizes the following rules regarding the ordering of statements:

- A PROGRAM statement, if present, or a SUBROUTINE, FUNCTION statement must be the first statement of the program unit.
- FORMAT statements may appear anywhere after the SUBROUTINE or FUNCTION, or PROGRAM statement.
- All specification statements must precede all DATA, statement function, and executable statements.
- All DATA statements must appear after the specification statements and before all statement function and executable statements.
- All function statements must precede all executable statements.
- Within the specification statements, the IMPLICIT statement must precede all other specification statements.

These statement ordering rules are illustrated in Figure 2-1.

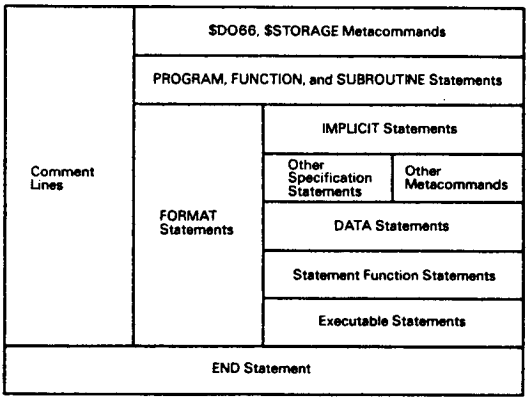

Figure 2-1. The Order of Statements within Program Units

Figure 2-1 demonstrates that

- PROGRAM, FUNCTION, and SUBROUTINE statements must go in your program first, followed by the IMPLICIT statements, other specification statements, DATA statements, statement function statements, and executable statements.
- Comment lines can go anywhere in the program.
- FORMAT statements can go anywhere in the program after the PROGRAM, FUNCTION, or SUBROUTINE statements.
- The END statement goes last in the program.

2.9 DATA TYPES

There are five basic data types: integer, single precision, double precision, logical, and character. This section describes the properties, the range of values, and the form of constants for each type.

2.9.1 Integer

Integer constants consist of a sequence of one or more decimal digits preceded by an optional arithmetic sign (+ or -). An integer cannot contain a decimal point. An integer variable occupies two or four bytes of memory depending upon the setting of the STORAGE metacommand. A 2-byte integer can contain any value in the range -32767 to 32767. A 4-byte integer can contain any value in the range -2, 147,. 483, 647 to 2, 147, 483, 647.

Integer constants consist of a sequence of one or more decimal digits or a radix specifier, followed by a string of digits in the range 0... (radix -1), where values between 10 and 35 are represented by the letters A through Z, respectively.

A radix specifier consists of the character "#", optionally preceded by a string of decimal characters that represent the integer value of the radix. If the string is omitted, the radix is assumed to be 16. If the radix specifier is omitted, the radix is assumed to be 10.

The following are examples of integer constants:

```
   123      +123      -123       0
 00123     32767    -32767
-#AB05  2#010111 -36#ABZ07
```

You can create an integer of a certain type by printing the following:

```
INTEGER*2
INTEGER*4
INTEGER
```

The first two commands specify 2 and 4 byte integers, respectively. The third command specifies either a 2 or 4 byte integer. The default value is 4, but you can use the STORAGE metacommand to allocate either 2 or 4 bytes. (For more information on the STORAGE metacommand, see Section 10.2.4.)

2.9.2 Single Precision Real

A real number (REAL or REAL*4) is a number containing a fractional part. In FORTRAN, a real number has either a decimal point, an exponent, or both. The compiler indicates the presence of the exponent with the letter E in the case of single precision real numbers, or the letter D in the case of double precision real numbers, followed by the exponent itself.

A single precision real value, occupying four bytes of memory, has a precision greater than six decimal digits. The range of single precision real values is approximately

```
 8.43E-37   to  3.37E+38 (positive range)
-3.37E+38   to -8.43E-37 (negative range)
 0
```

The precision is greater than six decimal digits. Single precision real numbers can take a number of different formats. Basic real constants consists of the following:

- An optional sign
- An integer part
- A decimal point
- A fractional part
- An optional exponent part

All of the following represent the same number -- one one-hundreth:

```
+1.000E-2          1.E-2          1E-2
+0.01            100.0E-4             .0001E+2
```

2.9.3 Double Precision

A double precision real number (REAL*8 or DOUBLE PRECISION) is a number containing a fractional part occupying eight bytes of memory. While the single precision real number has a precision greater than 6 decimal digits, the double precision real number has a precision greater than 15 decimal digits. The range of double precision real values is approximately

```
 4.19D-307  to   1.67D+308 (positive range)
-1.67+308   to  -4.19D-307 (negative range)
  0
```

A double precision constant consists of an optional sign followed by a real number which is in turn followed by the letter D and an exponent. The D distinguished the double precision number from a single precision number. A double precision constant consists of:

- An optional sign
- An integer part
- A decimal point
- A fractional part
- A required exponent beginning with D

A double precision constant is either a basic real constant followed by an exponent part or an integer constant followed by an exponent part. Some examples of double precision constants are as follows.

```
+1.123456789D-2        .00012345D+2       1.D-2
+0.000000001D0      100.0000005D-4     1D-2
```

The exponent can have an optional sign. The exponent indicates that the value preceding it is to be multiplied by ten to the value of the exponent's integer. If the exponent is zero, it must be specified as zero.

2.9.4 Logical

The logical data type consists of the two logical values: .TRUE. and .FALSE. A logical variable occupies two or four bytes of memory depending upon the setting of the STORAGE metacommand. LOGICAL*2 occupies two bytes. The least significant byte is either 0 (.FALSE.) or 1 (.TRUE.); the most significant byte is undefined. LOGICAL*4 occupies two words, the least significant of which contains the LOGICAL*2 value. The most significant is undefined.

2.9.5 Character

The character data type consists of a string of between 1 and 127 ASCII characters and spaces, occupying one byte per character or space. The length of a character value is equal to the number of characters in the sequence.

Character variables are assigned to contiguous bytes without regard for word boundaries. However, the compiler assumes that noncharacter variables that follow character variables always start on word boundaries.

A character constant consists of a sequence of one or more characters enclosed by a pair of single quotation marks.

```
'Joshua Clayborn'       'The blue lass'
```

Because apostrophes enclose character constants, you represent apostrophes themselves within the string by using a double apostrophe. This symbol prints as a single apostrophe and only occupies one byte of memory.

```
'Joshua''s pipe'        'Andrea''s house'
```

You can represent character constants that go over the 72 columns permitted in a FORTRAN line through the use of continuation lines. When a character constant extends across a line boundary, its value is as if the portion of the continuation line beginning with column 7 is appended to column 72 of the initial line. Thus, in the following example, the 58 blank spaces between the C and the D are equivalent to the space from C in column 15 to column 72 plus one blank in column 7 of the continuation line. You can represent very long character constants in this manner.

```
Example 1:  200    CH = "ABC
                  X  DEF'

Example 2:  200  CH = "ABC (58 blank spaces) ... DEF'
```

3

Expressions

CHAPTER 3
EXPRESSIONS

3.1 INTRODUCTION

An expression is a formula for computing a value. It consists of a sequence of operands and operators. The operands can contain function invocations, variables, constants, or other expressions. The operators specify the actions to be performed on the operands. FORTRAN has four types of expressions:

- arithmetic
- character
- relational
- logical

3.2 ARITHMETIC EXPRESSIONS

An arithmetic expression in FORTRAN involves constants and variables connected by arithmetic operators such as plus or minus signs. The simplest forms of arithmetic expressions are as follows:

- integer, real, or double precision constants
- integer, real, or double precision variable references
- integer, real, or double precision array element references
- integer, real, or double precision function references

The value of a variable reference or array element reference must be defined before it can appear in an arithmetic expression. Moreover, the value of an integer variable must be defined with an arithmetic value, rather than a statement label value previously set in an ASSIGN statement.

The order of operations in FORTRAN is the same as it is in algebra. Table 3-1 shows the operations in order of their precedence from highest to lowest order.

Table 3-1. Arithmetic Operators

Operator	Operation	Precedence
**	Exponentiation	Highest
/	Division	Intermediate
*	Multiplication	Intermediate
-	Subtraction or Negation	Lowest
+	Addition or Identity	Lowest

When an expression contains operations of different precedence, the operations occur in the order of their precedence. In the following example, B is first divided by C, and then the result is added to A:

```
A+B/C
```

When an expression contains operations of equal precedence, the operations to the left take precedence over those to the right. In the following example, A and B are added together first, and then C is subtracted from the result:

```
A+B-C
```

Exponentiation is the one operation that does not follow the left to right rule. Instead, the exponents to the right have precedence over those to the left. In the following example, B is taken to the C power, and A is taken to the resulting power.

```
A**B**C
```

When parentheses surround an operation, the operation within the parentheses takes precedence over those of a higher order. In the case of nested parentheses, the order of operations proceeds from the innermost parentheses to the outermost.

FORTRAN prohibits two operators from appearing consecutively. The following operation is illegal:

```
A**-B
```

You can use parentheses to separate operators:

```
A**(-B)
```

The operation above is legal and is interpreted as follows:

```
1/(A*B)
```

Unary minus is of the lowest precedence so that

```
-A*B
```

is interpreted as

```
-(A*B).
```

3.2.1 Integer Division

In FORTRAN, when you divide two integers, the operation produces an integer. The processor truncates any remainder downward toward zero. Thus, 7/3 evaluates to 2, and (-7)/3 evaluates to -2. Both 9/10 and 9/(-10) evaluate to zero.

3.2.2 Type Conversions

When all operands of an arithmetic expression are of the same data type, the expression produces the same data type. When the operands are of different data types, the expression produces the data type of the highest ranked operand. The ranks are illustrated in Table 3-2.

Table 3-2. Data Type Ranks

Data Type	Rank
Integer*2	1 (Lowest)
Integer*4	2
Real*4	3
Real*8	4

An operation on an integer and a real element produces a value of data type real. When an operation involves INTEGER*2 and INTEGER*4, the result is INTEGER*4. The data type of an expression is the data type of the result of the last operation performed in evaluating the expression.

You can perform integer operations on integer operands only. You can perform real operations on real operands or combinations of real and integer operands only. A fraction resulting from the division is truncated in integer arithmetic, not rounded. Thus, the following evaluates to zero, not one.

```
1/4 + 1/4 + 1/4 + 1/4
```

Real operations are performed on real operands or on combinations of real and integer operands only. The processor converts the integer to a real number by adding a decimal point followed by a zero. Real arithmetic is then used to evaluate the expression. In the following statement, integer division is performed on I and J, and a real multiplication is performed on the result multiplied by X.

```
Y=(I/J)*X
```

3.3 CHARACTER EXPRESSIONS

A character expression produces a value that is of type character. The forms of character expressions are as follows:

- character constant
- character variable reference
- character array element reference
- any character expression enclosed in parentheses

There are no operators that result in character expressions.

3.4 LOGICAL EXPRESSIONS

Logical expressions produce results that are either true or false. The simplest kinds of logical expressions are

- relational expressions
- logical variable references
- logical array element references
- logical function references
- logical constants

Other logical expressions are built up from the simple forms by using parentheses and the logical operators listed in Table 3-3.

3.4.1 The Relational Expression

Relational expressions compare the values of two arithmetic or two character expressions. The result of a relational expression is always a logical value of either .TRUE. or .FALSE. Table 3-3 shows the six relational operators and the operation they perform.

Table 3-3. Relational Operators

Operator	Operation
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

Relational expressions with arithmetic operands can have one operand of type integer and one of type real. In this case, the processor converts the integer operand to type real before it evaluates the expression.

Relational expressions with character operands compare the position of their operands in the ASCII collating sequence (see APPENDIX J). An operand is less than another if it appears earlier in the collating sequence. If you compare operands of unequal length, the processor considers the shorter operand as if it were extended to the length of the longer operand by adding of spaces.

3.4.2 Logical Expressions

Logical expressions use the logical operators .NOT., .AND., and .OR. .AND. and .OR. are binary in nature, comparing two values to determine whether the result is .TRUE. or .FALSE. With the .AND. operator, both must be true for the result to be true; with the .OR. operator, either must be true for the result to be true. .OR. is nonexclusive; there is no logical operator for exclusive OR. Table 3-4 illustrates the results of all possible .AND. and .OR. operations:

Table 3-4. .AND. and .OR. Operations

X	Y	.AND.	.OR.
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

The logical operator .NOT. operates on only one value at a time. If the value is true, the result is false; if the value is false, the result is true. Table 3-5 illustrates the use of .NOT.

Table 3-5. The Results of a Logical .NOT. Operation

X	.NOT.
TRUE	FALSE
FALSE	TRUE

There is an order of operations for logical operators as there is for arithmetic operations. Table 3-6 illustrates the order of precedence. .NOT. is of the highest precedence, .OR. is of the lowest, .AND. as in the middle.

Table 3-6. Logical Operators

Operator	Operation	Precedence
.NOT.	Negation	Highest
.AND.	Conjunction	Intermediate
.OR.	Inclusive disjunction	Lowest

As in algebra, operations precede from left to right. If two operations of equal precedence occur in one expression, the one on the left takes precedence. The following operations, therefore, bring these results:

```
A .AND.  B .AND.  C
```

is equivalent to

```
(A .AND.  B) .AND.  C
```

As an example of the precedence rules:

```
.NOT.  A .OR.  B .AND.  C
```

is interpreted the same as

```
(.NOT.  A) .OR.  (B .AND.  C)
```

.NOT. takes first precedence, .AND. takes second precedence, and .OR. takes lowest precedence.

Two .NOT. operators cannot be adjacent to each other, although

```
A .AND. .NOT. B
```

is an example of a legal expression with two adjacent operators.

Logical operators have the same meaning as in standard mathematical semantics with .OR. being nonexclusive. The following example evaluates to b .TRUE.:

```
.TRUE. .OR. .TRUE.
```

3.5 PRECEDENCE OF OPERATORS

When arithmetic, relational, and logical operators appear in the same expression, they abide by the precedence guidelines shown in Table 3-7.

Table 3-7. Relative Precedence of Operator Classes

Operator	Precedence
Arithmetic	Highest
Relational	Intermediate
Logical	Lowest

3.6 EVALUATING EXPRESSIONS

The following rules apply to the evaluation of expressions:

- The value of a variable, array element, or function reference must be defined before it can appear in an arithmetic expression.
- The value of an integer variable must be defined with an arithmetic value, rather than with a statement label value previously set in a ASSIGN statement.
- The following arithmetic operations are illegal:
  - dividing by zero
  - raising a zero-value operand to a zero or negative power
  - raising a negative-value operand to a power of type real

4

Assigning Value to Data

CHAPTER 4
ASSIGNING VALUE TO DATA

4.1 INTRODUCTION

Two statements, the DATA statement and the ASSIGNMENT statement give value to variables within your program.

4.2 THE DATA STATEMENT

The DATA statement is a nonexecutable statement that assigns initial values to variables. It must appear after all specification statements and prior to any statement function statements or executable statements. The format for DATA statements is as follows:

```
DATA list of variables/list of constants/[[,] list of variables/list of
    constants/]...
```

The list of variables includes variables, array elements, or array names; the list of constants includes constants or constants preceded by an integer constant repeat factor and an asterisk. The repeat factor causes the constant of a specified value to repeat as often as the repeat factor specifies. In the following examples, 3.14 repeats 5 times; help repeats 3 times, and 0 repeats 100 times:

```
5*3.14    3*'HELP'    100*0
```

The number of elements in the list of constants must match the number of elements in the list of variables. If an array appears in the list of variables, it is equivalent to a list of all the elements in that array in memory sequential order. You must index array elements only by constant subscripts.

The following DATA statement initializes A at 2.1 and I at 4:

```
DATA A/2.1/,I/4/
```

The DATA statement offers no data type information. The type of data is of the default type unless you precede the DATA statement with a type statement.

In the following example, the type statement declares the data as integer, while the DATA statement initializes the variables X2 and ZED with the values 5 and 7:

```
INTEGER X2,ZED
DATA X2,ZED/5,7/
```

You must declare the dimensions of an array before using the array name in the DATA statement. The following example declares two arrays, J and X, and sets the second element of J to equal 5, and both elements of X to equal zero. To set two elements of one array to equal the same number, the DATA statement uses the repeat element described above.

```
DIMENSION J(5),X(2)
DATA J(2),X/5, 2*0/
```

Consider the following rules when using the DATA statement:

- The DATA statement must appear after all specification statements, but before to any statement function or executable statement.
- The type of each noncharacter element in a list of constants must be of the same type as the corresponding variable or array element in the accompanying list of variables. For example, if the constant type is INTEGER*2, then the variable must be INTEGER*2. However, with the $NOTSTRICT metacommand in effect, a character element in the list of constants can correspond to a variable of any type.
- A character element in the list of constants must have a length that is less than or equal to the length of the variable or array element in the list of variables. If the length of the character constant is shorter, the compiler adds blank characters to the right.
- You cannot use a single-character constant to define more than one variable or array element.
- There must be the same number of values in each list of constants as there are variables or array elements in the corresponding list of variables.
- An array is equivalent to all of the elements in that array in memory sequence.
- You can index arrays only with constant subscripts.
- Only local variables and array elements can appear in a DATA statement.
- You cannot assign initial values through a DATA statement to formal arguments, variables in common, and function names.

4.3 THE ASSIGNMENT STATEMENT

An assignment statement assigns a value to a variable or an array element. There are two basic kinds of assignment statements: computational and label.

4.3.1 The Computational Assignment Statement

In the computational assignment statement, the processor evaluates the expression on the right and assigns the resulting value to the variable or array element on the left. In the following example, the processor evaluates 10/2 as 5 and assigns that value to L.

```
L=10/2
```

The syntax of a computational assignment statement is

```
variable = expression
```

The variable can either be a variable or an array element.

There are three types of computational assignment statements: arithmetic, logical, and character. In all three, the variable and the expression must be of the same data type. If the variable is a logical variable, for example, the expression must be a logical expression.

In an arithmetic assignment statement, if the types of the elements are not the same, the computer automatically converts the type of the expression to the type in the variable. When the expression is real, the compiler truncates the decimal part of the expression and assigns the expression to the integer variable. When the expression is an integer, the compiler adds a decimal point and a zero and assigns the expression to the real variable.

The rules for the conversion of expressions are given in Table 4-1.

Table 4-1. Type Conversion for Arithmetic Assignment Statements

Variable or Array	Integer	Real	Double Precision
Element(V)	Expression (E)	Expression (E)	Expression (E)
Integer	Assign E to V	Truncate E to integer and assign to V	Truncate E to integer and assign to V
Real	Append fraction (.0) to E and assign to V	Assign E to V	Assign most significant portion of E to V; least significant portion of E is rounded
Double Precision	Append fraction (.0) to E and assign to V	Assign E to most significant portion of V; least signif-icant portion of V is 0	Assign E to V

If the length of the expression does not match the size of the variable in a character assignment statement, the compiler adjusts it. If the expression is shorter, the compiler pads it with blanks to the right. If the expression is longer, the compiler truncates characters on the right.

You can assign logical expressions of any size to logical variables of any size without affecting the value of the expression. You cannot assign integer and real expressions to logical variables, nor can you assign logical expressions to integer or real variables. The compiler treats INTEGER*2 and INTEGER*4 identically except that it assigns INTEGER*4 in accordance with its range of values.

4.3.2 The Label Assignment Statement

The ASSIGN statement gives the value of a format or statement label to an integer variable. For example, if a GOTO statement follows the ASSIGN statement, control passes to the program statement that has the same label value as the variable in the ASSIGN statement. In the following example, control passes to the program statement labelled 3:

```
ASSIGN 3 to KIX
GOTO KIX
```

The format of the ASSIGN statement is

```
ASSIGN label to variable
```

The label is the label in the program, and the variable is an integer variabl

The ASSIGN statement sets the integer variable to the value of the label for either a format or statement label appearing in the same program unit. Once you assign the value of a label to a variable, the variable becomes undefined as an integer. You should not use it for any other purpose in the program. Errors at execution time can arise from such practices.

In the following example, you assign the label 10 to I. The GOTO statement commands that the processor pass command to the line assigned the label 10.

```
              ASSIGN 10 to I
              GOTO I
10            CONTINUE
```

5

Specification Statements

CHAPTER 5
SPECIFICATION STATEMENTS

5.1 INTRODUCTION

The term "specification statement" refers to a nonexecutable statement in Wang PC FORTRAN that defines the attributes of variables, arrays, and functions. For example, type statements declare data as being logical, integer, real, or character in nature. There are eight kinds of specification statements. Table 5-1 lists each specification statement and its function.

Table 5-1. The Function of Specification Statements

Statement	Purpose
COMMON	Provides for sharing memory between two or more program units.
DIMENSION	Specifies that a user name is an array and defines the number of its elements.
EQUIVALENCE	Specifies that two or more variables or arrays share the same memory.
EXTERNAL	Identifies a user-defined name as an external subroutine or function.
IMPLICIT	Defines the default type for user-defined names.
INTRINSIC	Declares that a name is an intrinsic function.
SAVE	Causes variables to retain their values across invocations of the procedure in which they are defined.
Type	Specifies the type of user-defined names.

Specification statements must precede all executable statements in a program unit but can appear in any order within their own group. The exception to this rule is the IMPLICIT statement. IMPLICIT statements must precede all other specification statements in a program unit.

5.2 THE IMPLICIT STATEMENT

Through the IMPLICIT statement, you declare the default type and size for variable names that begin with the letter(s) specified. The types and sizes are outlined in Table 5-2.

Table 5-2. Memory Requirements

Data Type	Bytes	Note
LOGICAL	2 or 4	1, 3
LOGICAL * 2	2	
LOGICAL*4	4	
INTEGER	2 or 4	1, 3
INTEGER*2	2 (Lowest)	
INTEGER*4	4	
CHARACTER	1	2
CHARACTER*n	n (maximum 127)	4
REAL	4	3, 5
REAL*4	4	
REAL*8	8 (Highest)	3, 6
DOUBLE PRECISION	8	

NOTES:
For integer and logical data, either 2 or 4 bytes are used. The default is 4 but may be set to either 2 or 4 with the STORAGE metacommand.

CHARACTER AND CHARACTER*1 are synonyms.

REAL and REAL*4 are synonyms.

The syntax of the IMPLICIT statement is

```
IMPLICIT type (letter [,letter]...) [,type (letter [,letter]...)]...
```

You can declare any one of the types and sizes in Table 5-2. The following rules apply to type.

- The default size is 4 bytes.
- Through the STORAGE metacommand, you can reset the default to 2. Refer to Subsection 10.3.4 for more information concerning the STORAGE metacommand.
- CHACACTER and CHARACTER*1 are synonyms.
- REAL and REAL*4 are synonyms.
- In CHARACTER*n, if the number of bytes (n) you specify is odd, the compiler uses one extra byte to begin on an even boundary.

- The letter must be a single letter or a range of letters. A range is indicated by the first and last letters in the range separated a minus sign.

The IMPLICIT statement below specifies that variables beginning with B are of character type and of length 6; variables beginning with E-G are of integer type and of length 2 or 4 by default (depending upon the setting of the STORAGE metacommand):

```
IMPLICIT CHARACTER*6(B), INTEGER (E-G)
```

IMPLICIT statements have the following characteristics:

- Explicit type statements override IMPLICIT statements when the two are in conflict. In the following example, the explicit REAL statement overrides the IMPLICIT statement.

```
IMPLICIT INTEGER (A-N)
REAL ANTON
```

- Explicit type in FUNCTION statements takes priority over an IMPLICIT statement.
- For character data, a subsequent type definition overrides the length of the user name.
- You cannot define the same letter with an IMPLICIT statement more than once in the same program unit.
- All IMPLICIT statements must precede all other specification statements in the program unit.
- An IMPLICIT statement applies only to the program unit in which it appears.
- IMPLICIT statements do not change the type of an intrinsic function.

5.3 THE DIMENSION STATEMENT

The DIMENSION statement specifies that a user name is an array and defines the number of its elements. The format of a DIMENSION statement is

```
DIMENSION array, (subscript(s))[,array (subscript(s))]...
```

The number of dimension declarators following an array indicates the number of dimensions in the array. The dimension declarators specify the dimensions of the array. The maximum number of dimensions is seven and conforms to the full language FORTRAN 77 standard. The maximum size of an array is 65,366 bytes. The maximum size of a dimension is 32,767 bytes.

In the following example, KIM possesses only one dimension with 30 elements; ALTER has two dimensions containing J and K elements; and A has three dimensions, one with five, one with six, and one with seven elements.

```
DIMENSION KIM(30)
DIMENSION ALTER (J,K)
DIMENSION A(5,6,7)
```

You can use any of the following specifiers as a dimension declarators:

- an unsigned integer constant
- a user name corresponding to a nonarray integer formal argument
- an asterisk

If a dimension declarator is an integer constant, the array has the corresponding number of elements in that dimension. An array has a constant size if all of its dimension specifiers are integer constants.

If a dimension declarator is an integer argument, that dimension is equal to the value of the argument upon entry to the subprogram unit at execution time. This array is an adjustable-sized array.

If the dimension declarator is an asterisk, the array is an assumed-sized array with an unspecified upper bound. Assumed-size dimension declarators can only appear as the last dimension in an array declarator. All adjustable and assumed-sized arrays must be formal arguments to the program unit in which they appear.

The processor stores single dimensional arrays in higher and higher memory locations and multidimensional arrays in linear columns. It stores X(4,2) as follows:

```
X(1,1) X(2,1) X(3,1) X(4,1) X(1,2) X(2,2) X(3,2) X(4,2)
```

The following dimension statements results in the following mapping, assuming A is placed at location 1000.

```
INTEGER*2 A (2,3)
DATA A /1,2,3,4,5,6/
```

Array Element	Address	Value
A (1, 1)	1000	1
A (2, 1)	1002	2
A (1, 2)	1004	3
A (2, 2)	1006	4
A (1, 3)	1008	5
A (2, 3)	100A	6

The processor reads and prints the elements of an array in the order in which it stores them.

5.4 ARRAY ELEMENT REFERENCES

The array declarator statement allows you to specify an array name, its dimensions, and its sizes within a program unit. Its format is as follows:

```
array (subscript [, subscript]...)
```

The subscript is an integer expression used in selecting a specific element in an array. The number of subscript expressions must match the number of dimensions in the array declarator. The value of a subscript expression must be between one and the upper limit for the dimension it represents.

In the following examples, HELEN is a 10 element single dimensional array. IAR is a two dimensional array, having 14 elements in the first dimension and 2 in the second dimension. The single dimensional array, NAME, reserves 12 bytes of storage for character data for each of its 24 elements.

```
HELEN(10)
IAR(14,2)
CHARACTER*12 NAME (24)
```

Through the use of the subscript, you can name any element in the array. For example, HELEN(3) names the third element in the array HELEN.

5.5 THE TYPE STATEMENT

A type statement is a declaration that tells the compiler what type of data and of what size to expect in a variable, array, external function, statement function or dimension: real, integer, logical, or character data. The type declaration statement has the following syntax:

```
type variable [*length specifier][,variable][*length specifier]...
```

Type refers to one of the types listed in Table 5-2; variable is the symbolic name of a variable, array, statement function, function subprogram, or an array declarator. You can specify a new length by appending a length specifier to the data type that you are declaring. This specifier must correspond to one of lengths indicated in Table 5-2 for the specific data type that you are using. If you use both an array declarator and a length declarator, put the length declarator last as follows:

```
INVEC(10)*4
```

The following rules apply to a type declaration statement:

- A type declaration statement must precede all executable statements.
- You can declare the data type of a symbolic name only once per program unit. The type declaration is in effect for the entire program unit.

- You cannot label a type declaration statement.
- You can use a type declaration statement to declare an array by appending a dimension declarator to an array name.
- A type statement confirms or overrides the implicit type of name.
- The name of a subroutine or main program cannot appear in a type statement.

A symbolic name can be followed by a data type length specifier of the form *length where length is one of the acceptable lengths for the data type being declared. This specification overrides the length attribute that the statement implies and assigns a new length to the specified item. If both a data type length specifier and an array declarator are included, the data type length specifier goes last.

In the following type statements, KATE and IABS are REAL variables, one byte; Q is a word long INTEGER variable; SWITCH is a byte long LOGICAL variable; NAME a ten byte long CHARACTER variable; and MATRIX is a two dimensional integer array.

```
REAL KATE, IABS
INTEGER*2 Q
LOGICAL SWITCH
CHARACTER*10 NAME
INTEGER MATRIX (4,4)
```

5.6 THE COMMON STATEMENT

A COMMON statement allows two or more program units to share memory. The COMMON statement tells the compiler to place a list of variables or arrays in a common area of memory where it is available to all program units. The COMMON statement eliminates the need for a parameter list in SUBROUTINE or CALL statements. The format of the COMMON statement is:

```
COMMON[/[common block name]/]list[[,]/[common block name]/list]...
```

There are two types of COMMON blocks, named and blank. In each COMMON statement, all variables and arrays appearing in each list of names following the common block name are in that common block. If you omit the first common block name, all of the elements in the list of names are in the blank common block. The list contains the names of variables, arrays, and array declarators separated by commas. It does not contain argument names or function names. In the following example, C and D belong to an unlabelled common area; E and F belong to a labelled common area, AREA1.

```
COMMON C,D/AREA1/E,F
```

The following rules apply to the COMMON statement:

- If a reference to a named common block occurs from several different program units, the common blocks must be the same length.
- The type and number of variables that appear in COMMON statements in different program units must correspond. The variable names within a common block, however, may vary between program units.
- Blank common blocks can have different lengths in different program units. The maximum length can occur in a program unit.
- Two variables that share a memory location must be aligned on even-byte boundaries.
- All variables in common areas are adjacent in memory in the order that you name them in the COMMON declaration.
- All elements in all lists for the same named common block are allocated in the common area in the order they appear in the list following the COMMON statement.
- Common block names can appear more than once in COMMON statements in the same program unit.
- If any element in a common area is of type character, all elements in that block must be of type character.
- If two program units refer to the same common block containing character data, they must be of the same length.
- The size of a common block is equal to the number of bytes required to hold all of its elements.

In the following example, I, J, X, and K(10) are part of an unlabelled common block; A(3) is part of the labelled common block, MYCOM. Through the use of the common block, you can call the subroutine MYSUB without using parameters.

```
PROGRAM MYPROG
COMMON I, J, X, K(10)
COMMON /MYCOM/ A(3)
.
.
.
END


SUBROUTINE MYSUB
COMMON I, J, X, K(10)
COMMON /MYCOM/ A(3)
.
.
.
END
```

5.7 THE EXTERNAL STATEMENT

The EXTERNAL statement indicates to the compiler that a given parameter is a symbol for a subprogram or function rather than for a variable. Because the FORTRAN compiler assumes that any undefined reference is external, you only use an EXTERNAL statement when passing the name of a subprogram or built-in function to another subprogram as one of its parameters. The format of an EXTERNAL statement is:

```
EXTERNAL name [,name]...
```

Name is the name of an external subroutine or function.

The following rules apply to the EXTERNAL statement:

- A user name can only appear once in an EXTERNAL statement in a given program unit.
- Statement function names cannot appear in an EXTERNAL statement.
- An EXTERNAL statement overrides the name of an intrinsic function. If you use an intrinsic function name in an EXTERNAL statement, the name becomes the name of an external procedure, and you can no longer call the intrinsic function from that program unit.

In the following example, the subprograms or functions MYFUNC AND MYSUB are parameters of CALC:

```
EXTERNAL MYFUNC, MYSUB
CALL CALC (MYFUNC, MYSUB)
```

5.8 THE INTRINSIC STATEMENT

In Wang PC FORTRAN, many built-in or intrinsic functions and routines are available to the user. For example, a routine that rounds off numbers or one that calculates square roots is available. 9.4.3 in this manual lists all the intrinsic functions available in Wang Professional Computer FORTRAN. An INTRINSIC statement declares that a user name is an intrinsic function. The format of an INTRINSIC statement is as follows:

```
INTRINSIC function name [,function name]...
```

Some rules to consider in using the INTRINSIC statement are as follows:

- Each user name can only appear once in an INTRINSIC statement.
- If a name appears in an INTRINSIC statement, it cannot appear in an EXTERNAL statement.

- All names used in an INTRINSIC statement must be system-defined INTRINSIC functions.

In the following example, SIN and COS are intrinsic parameters to CALC2:

```
INTRINSIC SIN, COS
X = CALC2 (SIN, COS)
```

5.9 The SAVE STATEMENT

The SAVE statement causes variables to retain their values across invocations of the procedure in which they are defined. Normally, the entity specified by the SAVE statement does not become undefined as the result of the execution of a RETURN or END statement. When the entity is used in a common block, however, it may become redefined or undefined in another program unit. After being saved, variables in the common block have defined values if the current procedure is subsequently re-entered.

Since in the current implementation, all common blocks and variables are statistically allocated, all common blocks and variables are saved automatically. Therefore, in practice, the SAVE statement has no effect.

The format of a SAVE statement is

```
SAVE /named common block/ [,/named common block/]...
```

The name is the name of the common block. In the following example, you save the common block MYCOM.

```
SAVE/MYCOM/
```

5.10 THE EQUIVALENCE STATEMENT

At times, you can use two or more names for the same variable. The EQUIVALENCE statement makes the two names synonomous. In other words, an EQUIVALENCE statement causes two or more variables or arrays to share the same memory location. The format of an EQUIVALENCE statement is

```
EQUIVALENCE (name, name) [,(name, name)]...
```

The names that follow the EQUIVALENCE statement are the names of two or more variables, arrays, or array elements that share the same memory location. In the following example, A, B, and C become synonyms for the same value:

```
EQUIVALENCE (A,B,C)
```

Use the following rules when you use an EQUIVALENCE statement:

- For two elements of type character to be associated, they must be the same length.
- If an array name appears in an EQUIVALENCE statement, it refers to the first element in the array.
- An EQUIVALENCE statement cannot cause the same memory location to appear more than once. The following EQUIVALENCE statement is illegal because it forces the variable R to appear in two distinct memory locations:

```
REAL R,S(10)
EQUIVALENCE (R,S(1)),(R,S(5))
```

- An EQUIVALENCE statement cannot cause the compiler to store consecutive array elements out of sequential order. The following EQUIVALENCE statement is illegal because it attempts to store the elements of R and S out of sequential order:

```
REAL R(10),S(10)
EQUIVALENCE (R(1),S(1)),(R(5),S(6))
```

- The elements that the EQUIVALENCE statement equates should be of the same data type. If the shared elements are of different types, the EQUIVALENCE does not cause an automatic type conversion.
- An EQUIVALENCE statement should not associate a character element with a noncharacter element so that the compiler allocates the noncharacter element on a odd byte boundary.

When you use EQUIVALENCE statements and COMMON statements together, several further restrictions apply:

- An EQUIVALENCE statement cannot cause two different common blocks to share common memory.
- Although an EQUIVALENCE statement can extend a common block by adding memory following the common block, it cannot add memory preceding the common block. In the following example, the EQUIVALENCE statement is illegal because it attempts to add memory before the start of the block:

```
COMMON /ABCDE/ R(10)
REAL S(10)
EQUIVALENCE (R(1),S(10))
```

- Extending a common block by an EQUIVALENCE statement must not cause its length to be different from that of the common block in other program units.

In the following example, the CHARACTER data, FIRST(1), becomes a synonym for the array NAME (1); NAME(21), for MIDDLE (1); and NAME (41) for LAST (1):

```
 CHARACTER NAME, FIRST, MIDDLE, LAST
 DIMENSION NAME(60), FIRST(20),
1          MIDDLE(20), LAST(20)
 EQUIVALENCE (NAME(1), FIRST(1)),
1          (NAME(21),MIDDLE(1)),
2          (NAME(41), LAST(1))
```

6

Control Statements

CHAPTER 6
CONTROL STATEMENTS

6.1 INTRODUCTION

Normally, the processor executes statements sequentially. Control statements allow you to change the order in which the processor executes statements in a FORTRAN program. Refer to Table 6-1 for a listing of control statements and their functions.

Table 6-1. Control Statements and Their Functions

Statement	Function
CALL	Calls and executes a subroutine from another program unit.
CONTINUE	Used primarily as a convenient way to place statement labels, particularly as the terminal statement in a DO loop.
DO	Causes repetitive evaluation of the statements following the DO, through and including the ending statement.
ELSE	Introduces an ELSE block.
ELSEIF	Introduces and ELSEIF block.
END	Ends execution of a program unit.
ENDIF	Marks the end of the series of statements following a block IF statement.
GOTO	Transfers control elsewhere in the program depending upon the kind of GOTO statement used (assigned, computed, or unconditional).
IF	Causes conditional execution of some other statement(s) depending upon the evaluation of the expression and the kind of IF statement used (arithmetic, logical, or block).
PAUSE	Suspends program execution until the RETURN key is pressed.

RETURN	Returns control to the program unit that called a subroutine or function.
STOP	Terminates program.

The CALL and RETURN statements are associated with subroutines, and are described in Chapter 9.

6.2 UNCONDITIONAL GOTO

The unconditional GOTO statement transfers control to the statement with the label that follows GOTO. The format of an unconditional GOTO statement is

```
GOTO label
```

The label is a statement label of an executable statement found in the same program unit as the GOTO statement. In the following example, control passes to the statement labeled 10:

```
      GOTO 10
        .
        .
        .
        .
10  CONTINUE
```

Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. A special feature, extended DO loops, does permit jumping into a DO block. Refer to Subsection 10.3.1, the DO66 Metacommand.

6.3 COMPUTED GOTO

The computed GOTO passes control to any of a number of labeled statements. The statement to which control passes depends upon the worth of the variable which comprises part of the statement. If the value of the variable is two, control passes to the first label in the list; (if the value of the variable is two) control passes to the second label and so on). The format of a computed GOTO statement is

```
GOTO (label [label] ...) [,] integer expression
```

The label is a statement label of an executable statement found in the same program unit as the GOTO statement. If there are fewer labels than the integer expression indicates, the computed GOTO statement serves as a CONTINUE statement. Otherwise, the next statement is the one labeled by the integer expression.

If, for example, the data is of two types, red and blue, and you assig 1 to red and 2 to blue, you could write the following computed GOTO statemer

```
GOTO(7,11), KOLOR
```

If the value of KOLOR is 1(red), control passes to the statement labeled 7. On 2(blue), control passes to the statement labeled 11.

Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. A special feature, extended range DO loops, does permit jumping into a DO block. Refer to Subsection 10.3.1 for more information on the DO66 Metacommand.

6.4 ASSIGNED GOTO

The assigned GOTO statement is actually made up of two statements, the assignment statement and the GOTO statement. The assignment statement gives the value of a label to a variable. (See Section 4.2, The Assignment Statement.) The GOTO statement causes control to transfer to the label with the same value as the variable. The format of an assigned GOTO statement is.

```
GOTO variable [[,] (label[, label]...)]
```

The variable must be an integer variable name; the label is a statement label of an executable statement found in the same program unit as the assigned GOTO statement. A runtime error is generated if the label last assigned to the variable in the optional list of labels is not present and the $DEBUG metacommand has been selected.

In the following program segment, the assignment statement assigns the value 5 to the variable KIM. The GOTO statement passes control KIM, the statement labeled 5:

```
    ASSIGN 5 TO KIM
    GOTO KIM, (3,5,7)
5   IF (J.EQ.K) GOTO 200
```

The statement label and the value of the variable must match, i.e., KIM must have a label matching that of the assignment statement.

A special feature, extended range DO loops, permits jumping into a DO block. Refer to Subsection 10.3.1, the DO66 Metacommand, for more information on this feature.

6.5 ARITHMETIC IF

The arithmetic IF causes evaluation of an arithmetic expression and selection of a label based on its value. If the value of the expression referenced in the statement is negative, control passes to the statement numbered by the first label in the sequence; if the value of the expression is zero, control passes to the second label; if the value of the expression is positive, control passes to the third label. The format of an arithmetic IF statement is:

```
IF (expression) label1, label2, label3
```

If the expression is an integer or real expression, the three labels are statement labels of executable statements found in the same program unit as the arithmetic IF statement. The same statement label can appear more than once among the three labels.

Transferring control into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. A special feature, extended range Do loops, permits jumping into a DO block.

In the following example, control passes to the statement labelled 3 if the value of Y is a negative number, to 5 if the value of Y is zero, and to 9 if the value of Y is a positive number:

```
IF(Y) 3,5,9
```

6.6 LOGICAL IF

The logical IF statement causes a logical expression to be evaluated. If the value of the expression is .TRUE., the statement is executed. If the value of the expression is .FALSE., the statement is not executed and control passes on to the next executable statement. The format of a logical IF statement is

```
IF (logical expression) statement
```

The statement is any executable statement except a DO, block IF, ELSEIF, ELSE, ENDIF, END, or another logical IF statement.

In the following example, if A equals B, control passes to the statement labeled 9. If A is greater than or lesser than B, control passes to the next statement, A=10+2.

```
      IF (A.EQ.B) GOTO 9
      A=10+2
9     END
```

6.7 BLOCK IF STATEMENTS

Through the BLOCK IF, you can execute a group of statements if the expression evaluates to .TRUE. or pass control to an ELSE, ELSEIF, or ENDIF statement at the same IF level if the expression evaluates to .FALSE.

There are three basic types of Block IF statements in Wang PC FORTRAN:

- The simple Block IF
- The Block IF followed by ELSEIF Block(s)
- The nested BLOCK IF followed by ELSE Block(s)

6.7.1 The Simple BLOCK IF Statement

The simple Block IF statement executes the block of statements if the logical value of the expression is true. Figure 6-1 shows the format of the Simple Block IF statement.

```
IF logical expression THEN

    Any number of statements to be
    executed if the value of the
    logical expression is .TRUE..

ENDIF
```

Figure 6-1. The Simple Block IF Statement

In the following example, J and K take on new values only if I is less than 10. Otherwise, control passes to the nonexecutable ENDIF statement and continues from there:

```
IF (I.LT.10) THEN
    J=2*10
    K=J+5
ENDIF
```

6.7.2 IF-ELSEIF Block

The Block IF statement offers a number of alternates depending upon whether the value of the original expression evaluates to be .TRUE. or .FALSE. If the value of the expression evaluates to be .TRUE., control passes to the statements following the THEN. If it evaluates to be .FALSE., control passes to the following ELSEIF THEN. The processor evaluates the ELSEIF THEN expression. If the value of the expression evaluates to be .TRUE., the processor executes the statements in this block. If it evaluates to be .FALSE., the processor passes control to the next ELSEIF THEN and so on.

The Block IF-ELSEIF can include a final ELSE statement to which control passes when the value of all other logical statements evaluate to be .FALSE.. The Block IF-ELSEIF ends with an ENDIF statement. Figure 6-2 illustrates the format of the Block IF-ELSEIF.

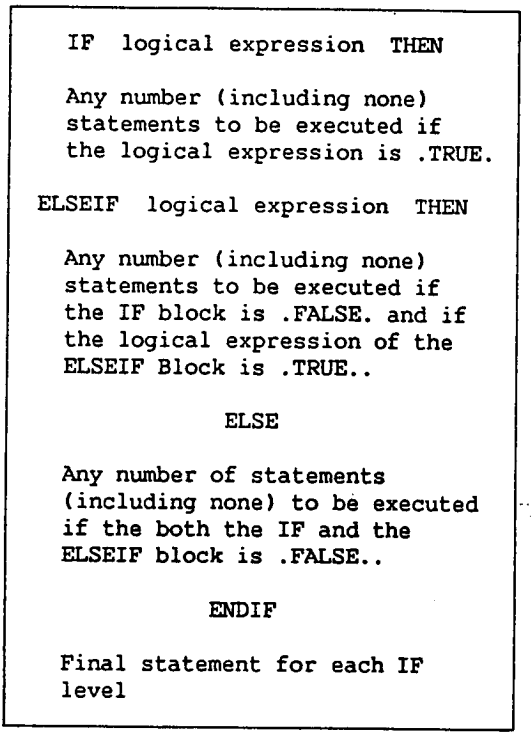

Figure 6-2. The IF THEN, ELSEIF Block

In the following program, K is given the value of 10 if J is greater than 1000; K is given the value of 8 if J is more than 100; K is given the value of 5 if J is more than 10. If J has a value of less than 10, K is given the value of 1.

```
IF (J.GT.1000)THEN
    K=10
ELSEIF (J.GT.100)THEN
    K=8
ELSEIF (J.GT.10)THEN
    K=5
ELSE
    K=1
ENDIF
```

6.7.3 Nested IF THEN ELSE BLOCKS

You can achieve results similar to those of the IF ELSEIF THEN Blocks through nested IF THEN statements followed by ELSE . A series of IF statements can set up a series of tests as to whether the value of a set of Block IF expressions are .TRUE.. If the value of the IF expressions are .TRUE., the processor executes them; if not, it does not.

ELSE statements can follow the IF statements. The processor executes the ELSE Block statements when the IF value of the expression is .FALSE.

The format of the nested IF THEN ELSE BLOCK is demonstrated in Figure 6-3.

```
IF  logical expression  THEN

    Any number (including none)
    statements to be executed if
    the value of the logical
    expression is .TRUE..

IF  logical expression  THEN

    Any number (including none)
    statements to be executed if
    the value of the logical
    expression is .TRUE..

IF  logical expression  THEN

    Any number (including none)
    statements to be executed if
    the value of the logical
    expression is .TRUE..

            ELSE

    Any number (including none)
    statements to be executed if
    the value of the logical
    expression is .FALSE..

            ELSE

    Any number (including none
    statements to be executed if
    the value of the logical
    expression is .FALSE..

           ENDIF

    Final statement for each IF level.

           ENDIF

    Final statement for each IF level.
```

Figure 6-3. The Nested Block IF Statement

The following program includes three nested IF THEN statements and two nested ELSE statements. The processor executes all of the IF statements if the expressions evaluate to be .TRUE.. It only executes the first ELSE statement if none of the IF THEN statements evaluate to be .TRUE. and if (L.EQ.M) evaluates to be .TRUE.. It only executes the second ELSE statement if all of the other IF and ELSE expressions evaluate to be .FALSE.

```
IF(L.GT.M) THEN
    J=10
    B=2.4
IF(L.LE.M) THEN
    J=J+2
    B=0
IF(L.EQ.N)THEN
    J=J+M
    B=L
ELSE
    If(L.EQ.M)THEN
        J=20
    ELSE
        J=100
    ENDIF
ENDIF
```

This program includes only one statement following each IF THEN, ELSEIF THEN or ELSE statement. However, you can add any number of statements. The number is restricted only by the size of the memory of your Professional Computer.

6.7.4 IF THEN, ELSEIF THEN, ELSE and ENDIF

Four separate statements are associated with the Block IF:

- IF THEN
- ELSEIF THEN
- ELSE
- ENDIF

The BLOCK IF and ELSEIF THEN

The Block IF statement causes the processor to evaluate the logical expression. If the value of the Block IF statement evaluates to be .TRUE., the processor executes the statements in the IF Block; if the value of the Block IF statement evaluates to be .FALSE., the processor passes control to the next IF, ELSEIF THEN or ELSE Block. If there are no other blocks, control passes to the ENDIF statement.

If ELSEIF THEN blocks exist in the program, the processor evaluates them in the order of their occurrence. If the value of the expressions within them evaluate to be .TRUE., the processor executes the statement(s) in the block. If the value of the expressions evaluate to be .FALSE., the processor passes control to the next ELSEIF THEN block, to the next ELSE block, or to the ENDIF block. If there are no other blocks in the program, execution passes to the ENDIF statement. The syntax of the Block IF statement and the ELSEIF statement is

```
IF logical expression THEN
    .
    .
    .
ELSEIF logical expression THEN
```

The ELSE statement

Control passes from IF or ELSEIF statements to ELSE statements if the value of the expressions in the IF or ELSEIF statements that precede the ELSE statement evaluate to be .FALSE.

The format of the ELSE statement is:

```
ELSE
```

An ENDIF statement must accompany each ELSE statement. The matching ENDIF statement must appear before any intervening ELSE or ELSEIF statements of the same IF level. Transfer of control into an ELSE block from outside that block is not permitted.

The ENDIF STATEMENT AND THE IF LEVEL

When the processor evaluates all parts of the IF block and completes all execution, control passes to the nonexecutable ENDIF statement. The ENDIF statement indicates that the IF block is complete. The format of a ENDIF statement is

```
ENDIF
```

Block IF statements require as many ENDIF statements as there are IF levels. The IF level is the number of block IF statements from the beginning of the program unit less the number of accompanying ENDIF statements. For example, if there were three IF statements in the program and one ENDIF statement, the IF level is two. The following rules apply to the IF level.

- The IF level of every statement must be greater than or equal to zero.
- The IF level of every IF, ELSEIF, ELSE, or ENDIF statement must be greater than zero.

- The IF level of every END statement must be zero.
- One IF statement with its accompanying logical expression determine one IF level.

The following program demonstrates the number of IF levels of a routine. There is one IF level for each of the first two IF THEN statements and one for the IF THEN statement in the ELSE Block:

```
IF (J.GT.K) THEN
    IF (J.LE.20) THEN
        J=J+K
    ENDIF
ELSE
    IF(J.EQ.20)THEN
      K=L+J
    ENDIF
ENDIF
```

6.8 DO

The DO statement repeatedly evaluates the statements following the DO through and including the statement with the indicated label. A DO statement has a range, beginning with the statement that follows the DO statement and ending with a the statement that the DO statement label references. The labeled statement is called the object of the DO or the terminal statement. It must follow the DO statement and be in the same program unit.

The labelled statement must not be an unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSEIF, ELSE, ENDIF, RETURN, STOP, END, or DO statement. If the terminal statement is a logical IF, it can contain any executable except those not permitted inside a logical IF statement.

Each DO statement needs a counter or an index which the processor increments each time it repeats the loop. It also needs an initial value for the index as well as a final value after which the processor passes control to other code. Finally, the DO statement needs an increment value that tells by how much the processor should increment the index with each repetition. The default value of the increment is 1. The syntax of a DO statement is:

```
DO label[,] index=initial value, test value [,increment value]
```

The label indicates the label of the object of the DO loop; the index is the counter for the DO loop; the initial value is the value of the index before the first repetition; the test value is the final value, after which the processor exits from the loop; and the increment value is the amount that the processor increments the counter upon each repetition of the loop.

The initial, test, and increment values can either be integer constants or integer variables. Either of the following DO statements is legal.

```
DO 101, IND = 3, 15, 3

DO 200, J = J, M
```

The DO statement controls the statements in its range by:

- Setting the index equal to the initial value.
- Executing the statements within the DO range.
- At the terminal statement adding the increment value to the index.
- Comparing the index to the terminal value.
- If the index is less than or equal to the terminal value, transferring control back to the first statement in the range.
- If the index value is more than the terminal value, passing control to the next executable statement following the terminal statement.

The following program prints the square root of the even integers from 2-20. The index is I; the label is 14; the initial value is 2; the terminal value is 20; and the increment value is 2.

```
    DO 14, I=2,20,2
14  WRITE I, I*I
    STOP
    END
```

The following rules apply to DO loops:

- You can nest DO statements, but the range of the nested statements must be contained within the range of the enclosing DO loop. DO loops can share terminal statements.
- The range of a DO loop must be entirely contained within an IF, ELSEIF, or ELSE block.
- An IF block statement must be entirely contained within the DO loop.
- The DO variable index may not be changed in any way by the statements within the range of the DO loop.
- Jumping into the range of a DO loop from outside its range is illegal although PC FORTRAN has a special feature added for compatibility with earlier versions of FORTRAN that permits "extended range" DO loops. Refer to Subsection 10.3.1 for more information concerning the $DO66 metacommand.

6.8.1 The Implied DO

Implied DO statements are often associated with READ and WRITE statements. They cause the associated statements to execute as many times as specified. Implied DO statements have the following format:

```
READ or WRITE (I/O list, integer variable=expression, expression,
[,expression])
```

The list is a variable name, an array element name, or an array name. The expressions are the same as those defined for the DO statement above. In the following example, the READ accompanied by the implied DO causes the processor to read the values of an array, G, for I ranging from 1 to 8:

```
READ, (G(I),I=1,8)
```

In a READ statement, the DO integer variable or an associated entity must not appear as an input list item in the embedded input/output list. It must but can have been read in the same READ statement before the implied DO list. The embedded input/output list is repeated for each iteration of the integer variable with appropriate substitution of values for the DO variable. The innermost loop of implied DO loops is always executed first.

6.9 CONTINUE

The CONTINUE statement causes no operation to be performed. It is used primarily as a terminal statement in a DO loop and as a convenient statement for labels. Execution of a CONTINUE statement has no effect upon the program. It marks the end of a DO loop and allows control to pass to the next executable statement. The syntax of a CONTINUE statement is:

```
CONTINUE
```

In the following example, the ten elements of IARRAY are initialized with the value 0. The CONTINUE statement marks the termination of the DO loop.

```
        DO 10, I = 1, 10
        IARRAY(I) = 0
10      CONTINUE
```

6.10 STOP

The STOP statement causes the program to terminate. After the command, you can write an optional message which displays on the terminal. The syntax of a STOP statement is:

```
STOP [message]
```

e message is a character constant or a string of not more than five digits.

In the following example, if IERROR is not equal to zero, the program erminates on the STOP statement with the message 'ERROR DETECTED':

```
        IF (IERROR .EQ. 0) GOTO 200
        STOP 'ERROR DETECTED'
200     CONTINUE
```

.11 PAUSE

The PAUSE statement causes the processor to suspend the program until it receives an indication from the keyboard to continue. If you have written a essage, it displays on the screen as a prompt requesting input from the eyboard. If you have not included your own message, the following messages isplays: 'PAUSE. Please press RETURN to continue.' To continue the rogram, press the RETURN key.

The format of a PAUSE statement is:

PAUSE [message]

The message is either a character constant or a string of not more than five igits.

In the following example, the processor pauses and displays the message WARNING: IWARN IS NONZERO' if the value of IWARN does not equal zero.

```
        IF (IWARN .EQ. 0) GOTO 300
        PAUSE 'WARNING: IWARN IS NONZERO'
300     CONTINUE
```

.12 END

The END statement terminates execution of the main program. It must appear as the last statement in every program unit. In a subprogram, the END statement has the same effect as a RETURN statement.

The format of an END statement is:

END

An END statement must appear on an initial line and contain no continuation lines. No other FORTRAN ending statement, such as the ENDIF statement, can have an initial line that appears to be an END statement.

The following END statement terminates program MYPROG:

```
        PROGRAM MYPROG
        WRITE(*, '(10H HI WORLD!)')
        END
```

7

I/O System

Records
Files
I/O Statements

CHAPTER 7
I/O SYSTEM

7.1 RECORDS

The building block of the FORTRAN file system is the record. A record is a sequence of characters or values. There are three kinds of records:

- formatted
- unformatted
- endfile

A formatted record is a sequence of characters that the processor converts into the type of data specified in a FORMAT statement: integer, character, real, and logical. Each formatted record is termined by the character value of the RETURN key. When reading a character variable, the processor converts the character values into numeric, logical, or character values, and when writing a character variable, the processor converts the values into their external character representation.

An unformatted record is a sequence of values that the processor neither interprets nor alters. No physical representation exists for the end of record. You do not specify the form in which the data is stored.

After the last record in a sequential file, the FORTRAN file system simulates a virtual endfile record. This prevents the processor from reading more information than is actually in the file. In PC FORTRAN, the endfile character is the ASCII CONTRL/Z.

7.2 FILES

A file is a sequence of records. Files are either external or internal.

- An external file is a collection of values written on I/O medium such as cards, magnetic tape, or a disk.

- An internal file is a character variable or character array element that is the source or destination of some I/O action. The file has exactly one record of the same length as the character variable or character array element. If you write less than the entire record, the processor fills the remaining portion of the record with blanks. The file position is always at the beginning of the file prior to execution of the I/O statement.

Through internal files, the processor converts values to and from their external character representations. When the processor reads a character variable, it converts the character values into numeric logical, or character values. When the processor writes a character variable, it converts the values into their external character representation. Only formatted, sequential I/O is permitted to internal files and only the I/O statements READ and WRITE can specify an internal file. You cannot use the backslash edit descriptor with internal files.

7.2.1 File Properties

A FORTRAN file has the following properties:

- name
- position
- formatted or unformatted
- sequential or direct access
- old and new

File Name

The file name of a program in PC FORTRAN follows the rules of the PC operating system. It is composed of up to eight characters followed by a period and an extension of up to three more characters. For example, the following is a legal FORTRAN name:

```
PROG1.FOR
```

File Position

The previous I/O operation usually sets up the position of a file. A file has an initial point, terminal point, current record, preceding record, and next record. It is possible to be between records in a file, in which case the next record is the successor to the previous record, and there is no current record.

When a sequential file is opened for writing, the file is positioned at its beginning, and all old data in the file is discarded. At the end of a sequential write, the file is positioned at the end of the file, but not beyond the endfile record. Once the processor has executed the ENDFILE, the file is positioned beyond the endfile record. A READ statement executed at the end of the file also positions the file beyond the endfile record. If you want to know what the endfile condition is, use the END=option in a READ statement.

File Structure

An external file can be opened as a formatted, unformatted, or binary file. All internal files are formatted. An internal file is a character variable or character array element. The file has exactly one record of the same length as the character variable or array element of which it is composed. The three file structures are as follows.

- Formatted files consist entirely of formatted records while.
- Unformatted files consist entirely of unformatted records.
- A binary file is a sequence of bytes with no internal structure.

Sequential and Direct Access Properties

An external file is either sequential or direct. Sequential files contain records whose order is determined by the order in which the records were written (the normal sequential order). In the READ or WRITE statement, you cannot use the REC= option to read or write sequential files.

Since direct access files (also called random access files) reside on a disk, the processor can read or write the records of direct access files in any order. The processor numbers each record with a unique number, specified when the record is written. All records have the same length specified when the file is written.

Records can be written out of order. For example, records 9, 5, and 11 can be written in that order without the records in between. Once written, a record cannot be deleted, but it can be overwritten with a new value. If you attempt to write a record beyond the old terminating file boundary, and there is more room on your device, the operating system extends the file. If you attempt to read a record from a direct access file that has not been written, an error is generated.

There are two kinds of devices that correspond to the two kinds of files: sequential and direct. The files associated with sequential devices are streams of characters; except for reading and writing, no explicit motion is allowed. The keyboard, screen, and printer are sequential devices.

Direct devices, such as disks, can access specific locations either sequentially or randomly, and thus support direct files. The PC FORTRAN file system does not allow direct files on sequential devices.

Old and New Files

An old file is a previously opened file. With disk files, opening a "NEW" file creates a new file. If you close that file or if your program terminates without a CLOSE on that file, a permanent file is created. If a previous file existed with the same name, the old file is deleted.

When you open a device such as the keyboard or the printer as a file, it makes no difference whether you use "OLD" or "NEW". With disk files, however, opening a file as NEW creates:

- A new file and deletes any previous file with the same name.
- A permanent file of the same name given when the file was opened if the new file is closed with STATUS='KEEP' or if the program terminates without a CLOSE operation.

There is no concept in FORTRAN of "opened for reading" as opposed to "opened for writing". You can open existing files and write to them; in this case, you overwrite them, and the data that they previously contained is lost. You can alternately write and read to the same file.

A WRITE to a sequential file deletes any records that existed beyond the newly written record.

7.2.2 Units

A unit is a means of referring to a file. A unit specified in an I/O statement is either an internal file specifier or an external unit. An internal file specifier is a character variable or character array element that directly specifies an internal file.

An external unit specifier stands for an external device such as a printer, a card reader, or a screen. The specifier itself is either a positive integer expression, or an asterisk (*) which stands for the screen for writing and the keyboard for reading.

The OPEN statement binds the value of the external unit specifier to a physical device or to external, sequential, or formatted files resident on the device. The external unit specifier value is uniquely associated with a particular external entity until an explicit CLOSE occurs or until the program terminates.

The asterisk represents the keyboard and screen; the character * is interpreted as unit zero. Unit zero requires no explicit OPEN statement since it is implicitly associated with the keyboard for reading and the screen for writing.

7.2.3 Commonly Used File Structures

Numerous combinations of file structures are possible in PC FORTRAN. The two most common types are:

- * files
- named, external, sequential, formatted files

The asterisk represents the keyboard and screen, a sequential, formatted file also known as unit zero. When reading from unit zero, you must enter an entire line. An external file can be bound to a system name by any of the following methods.

- You can specify the name in the OPEN statement.
- If you specify the name as all blanks in the OPEN statement, the name is read from the command line. If there is no command line or if no name exists, the processor prompts you for the name.
- If the file is opened implicitly with a READ or WRITE statement, the name is obtained as in the explicitly opened statement above.
- If you open the file explicitly and specify no name in the OPEN statement, the file is considered a scratch or temporary file and a default name is used.

7.2.4 Other File Structures

The less commonly used file structures are appropriate for certain classes of applications. A very general indication of the intended usages for them follows.

- If the I/O is random access, as would probably be the case in a data base, direct access files are necessary.
- If the data is written by FORTRAN and reread by FORTRAN, unformatted files are perhaps more efficient in terms of speed, but possibly less efficient in terms of disk space. The combination of direct and unformatted files is ideal for a data base which FORTRAN creates, maintains, and accesses.
- If the data must be transferred without any system interpretation, especially if all 256 possible byte values are to be transferred, unformatted I/O is necessary. One use of unformatted I/O is in the control of a device that has a single-byte, binary interface. Formatted I/O would interpret certain characters such as the ASCII representation for RETURN rather than passing them through to the program unaltered. The number of bytes written for an integer constant is determined by the $STORAGE metacommand.

- If the data needs to be transferred with no system interpretation, but will be read by non-FORTRAN program, the BINARY format is recommended. Binary files, a sequence of uninterpreted binary bits, contain only the data written to them. You cannot backspace them; you cannot read an incomplete record from them. For a non-FORTRAN program to interpret this data correctly, it must be compatible with this format.

7.3 I/O STATEMENTS

There are seven I/O statements: OPEN, CLOSE, READ, WRITE, BACKSPACE, ENDFILE, and REWIND. These statements take certain parameters and arguments that specify sources and destinations of data transfer and other facets of the I/O operation. Figure 7-1 describes the function for each of the I/O statements.

Table 7-1. The Function of the Input/Output Statements

Statement	Function
BACKSPACE	Positions the file connected to the specified unit to the beginning of the previous record.
CLOSE	Disconnects the unit specified and prevents subsequent I/O from being directed to that unit.
ENDFILE	Writes an end-of-file record on the file connected to the specified unit.
OPEN	Associates a unit number with an external device or with a file on an external device.
READ	Transfers data from a file to the items in an input/output list.
REWIND	Repositions a specified unit to the first record associated with the file.
WRITE	Transfers data from the items in an input/output list to a file.

Most I/O statements take unit specifiers. A unit specifier can be one of the following:

- * refers to the keyboard or screen.
- An integer expression which refers to an external file with a unit number equal to the value of the expression (* is unit number zero).
- The name of a character variable or character array element which refers to the internal file specified by the value of the variable array element

The format specifier in an I/O statement can be one of the following:

- A statement label which refers to the FORMAT statement.
- An integer variable name which refers to the FORMAT label assigned to integer variables by the ASSIGN statement.
- A character expression which refers to the format specified in the current value of the character expression provided as the format specifier.
- * indicates a list-directed I/O transfer.

The input/output list specifies the entities whose values are transferred by READ and WRITE statements. An input/output list can be empty, but it ordinarily consists of input or output entities and implied DO lists, separated by commas. An input entity can be specifies in the input/output list of a READ statement, and an output entity in the input/output list of a WRITE statement.

- An input entity is either a variable name, an array element name, or an array name. An array name specifies all of the elements of the array in memory sequence order.
- An output entity can be an array element name, an array name, or any other expression not beginning with a left parenthesis since the left parenthesis distinguished implied DO lists from expressions. To distinguish it from an implied DO list, you can use a plus sign in front of an expression as follows.

```
+(A+B) * (C+D)
```

7.3.1 The OPEN Statement

The OPEN statement binds a unit number with an external device or file on an external device. There are two kinds of OPEN statements: explicit and implicit statements. The explicit OPEN statement names the file directly; the implicit OPEN statement, part of the READ statement, opens a file without an explicit OPEN statement.

The syntax of an OPEN statement is:

```
OPEN unit number [,FILE=filename] [,STATUS="OLD" or "NEW"]
    [,ACCESS="SEQUENTIAL" or "DIRECT"][,FORM="FORMATTED" or
    "UNFORMATTED"],RECL=record length)
```

The unit number is an external unit specifier, a device number, that must appear as the first argument.

The file name is an optional character expression. If it is not present, the processor creates a temporary scratch file which you delete when you close the file. The file name appears as the second argument after the OPEN statement.

The STATUS is an optional parameter and is either 'OLD' (the default) or 'NEW.' 'OLD' is for reading or writing existing files. 'NEW' is for writing new files.

The ACCESS is an optional parameter and is either 'SEQUENTIAL' (the default) or 'DIRECT'.

The FORM is an optional parameter and is either 'FORMATTED' (the default) or'UNFORMATTED'.

RECL is a required argument with direct files only. The RECL=record length option is as integer expression that specifies the length of the records in that file.

The following example OPENS the new, sequential file FNAME and binds it to unit 7:

```
OPEN (7, FILE=FNAME, ACCESS='SEQUENTIAL', STATUS='NEW')
```

You can open a file with an implicit OPEN operation by using READ or WRITE statements. The implicit OPEN is equivalent to the following command:

```
OPEN (unit number, FILE= ' ',STATUS='OLD'
       ACCESS='SEQUENTIAL'
           FORM='FORMATTED', 0) if the read statement is formatted or
               'UNFORMATTED', 0) if the read statement is unformatted.
```

The following READ statement is an implicit OPEN statement. The unit number is unit zero, the ACCESS is SEQUENTIAL, and the FORM is FORMATTED in a character format represented by 'A'.

```
READ(0,'(A)') FNAME
```

Since unit zero is permanently connected to the keyboard, binding unit zero to a file has no effect.

The device numbers assigned to specific devices are:

0 Keyboard
1 Screen
2 Error
3 Communications part (Aux)
4 Printer

7.3.2 The CLOSE Statement

CLOSE disconnects the unit specified and prevents subsequent I/O from being directed to that unit.

The format of a CLOSE statement is:

```
CLOSE(unit number, [STATUS='KEEP'or'DELETE'])
```

The unit number is the external unit specifier, the device number.

The optional STATUS argument applies only to files that are apended NEW. The default is KEEP. If you indicate STATUS='DELETE', the processor discards the file. Normal termination of a FORTRAN program automatically closes all open files as if a STATUS = 'KEEP' were specified.

CLOSE for unit zero has no effect, since the CLOSE operation is not meaningful for keyboard and screen. The following statement closes and deletes the file bound to unit 7:

```
CLOSE(7,STATUS='DELETE')
```

CLOSE does not cause an END-OF-FILE record to be written.

7.3.3 The READ Statement

The READ statement causes the processor to bring information from the file connected to the external specifier into the variables in the variable list. The syntax of a READ statement is:

```
READ(unit specifier [,format statement number] [,REC=record number]
    [,END=label] [,ERR=label]) list of variables
```

The unit specifier is the external unit number of a connected file. The unit specifier is required and must appear as the first argument.

The format statement number is a label for the format statement and is required for the formatted READ as the second argument; it must not appear in the unformatted READ.

The REC=record number is for direct access files only. The record number, a positive integer expression, gives the record number of the record to be processed. If REC=record number is omitted for a direct access file, reading continues sequentially from the current position in the files.

END=label tells the line number in the program to which the processor is to go if it detects the end of the file during the READ operation. END=label is an optional statement. If it is not present, reading the end of the file results in a runtime error. If it is present, encountering an end of file condition causes control to transfer to the labeled statement. This statement must be in the same program unit as the READ statement.

ERR=label tells the line in the program to which the processor is to go in case an error condition occurs during the I/O operation. ERR=label is optional. If it is not present for a direct access file, an I/O error results in a runtime error. If it is present, I/O errors cause control to transfer to the labeled statement.

The READ statement brings information from the file connected to the external specifier into the list of variables. If the read is internal, the character variable or character array element specified is the source of the input; otherwise, the external unit is the source.

The following is a formatted READ statement with the format line at label 5000. The processor reads from device 2. If an end of file condition is encountered during the READ operation, control switches to the line labeled 120. If the processor detects an error condition, control passes to the line labeled 300. The processor reads the values into the variables A, B, C.

```
READ (2,5000, END=120, ERR=300) A, B, C
```

7.3.4 The WRITE Statement

The WRITE statement sends information from the variables in the variable list to the file connected to the file specifier. The syntax of a WRITE statement is:

```
WRITE(unit specifier [,format label][,ERR=label][,REC=record
    number])variable list
```

The unit specifier is required and must appear as the first argument.

The format label is required for the formatted WRITE as the second argument; it must not appear for unformatted write.

ERR=label is an optional statement label. If it is not present, I/O errors result in runtime errors. If it is present, I/O errors cause control to transfer to the labeled executable statement.

REC=record number is for direct access only; otherwise, an error results. It is a positive integer expression. It gives the record number for the record to be processed. If you omit REC=record number for a direct access file, writing continues from the current position in the file.

In the following example, the command causes information in THERM to be written into the file connected with 9. The statement labeled 2100 contains the format information. The record number is IB.

```
WRITE(9, 2100, REC=IB) THERM
```

7.3.5 The BACKSPACE Statement

BACKSPACE causes the file connected to the specified unit to be positioned before the preceding record. If there is no preceding record, the file position does not change. If the preceding record is the endfile record, the file becomes positioned before the endfile record. If the file position is in the middle of the record, BACKSPACE repositions to the start of that record. The syntax of a BACKSPACE statement is:

```
BACKSPACE unit specifier
```

The unit specifier cannot be an internal unit specifier. In the following example, the device stipulated by 1 backspaces:

```
BACKSPACE 1
```

7.3.6 The ENDFILE Statement

ENDFILE places an end-of-file record in the file. You cannot transfer further sequential data until you execute a BACKSPACE or REWIND. An ENDFILE on a direct access file makes all records written beyond the position of the new end-of-file become deleted. The syntax of an ENDFILE statement is:

```
ENDFILE unit specifier
```

The unit specifier must be an external unit specifier.

The following example causes the processor to place an end-of-file in the file associated with 1:

```
ENDFILE1
```

7.3.7 The REWIND Statement

Execution of a REWIND statement causes the file associated with the specified unit to be repositioned at its initial point. The syntax of a REWIND statement is:

```
REWIND unit specifier
```

The unit specifier must be an external unit specifier. The following statement causes the unit associated with 2 be repositioned at the beginning of the file:

```
REWIND2
```

8

The Format Statement

CHAPTER 8
THE FORMAT STATEMENT

8.1 THE FORMAT STATEMENT

FORMAT statements are always associated with input/output statements. The FORMAT statement describes the external appearance of the input or output record. The format of the FORMAT statement is:

```
statement label FORMAT ([descriptor1, descriptor2,...])
```

Up to three levels of nested parentheses are permitted within the outermost level of parentheses. You may omit the comma between the two list items if the resulting format specification is not ambiguous, i.e., after a P edit descriptor or before a slash edit descriptor.

FORMAT statements must be labeled and cannot be the target of a branching operation.

Formatted READ or WRITE statements must have accompanying FORMAT statements. These FORMAT statements can either be explicit or implicit. An explicit FORMAT statement is a formal FORMAT statement referenced by a READ or WRITE statement. The following WRITE statement references the explicit FORMAT statement labeled 990:

```
    WRITE (*, 990) J,A,K
990 FORMAT (I5,F5.2,I3)
```

The implicit FORMAT statement references the format within the WRITE or READ statement itself. The following WRITE statement is an equivalent means of specifying the same format as in the example above:

```
WRITE (*, '(I5,F5.2,I3)' J,A,K
```

The third way of referencing a FORMAT statement is through an ASSIGN statement as follows.

```
    ASSIGN 990 to IFMT
990 FORMAT (I5)
    WRITE (*, IFMT)
```

A fourth way of referencing a FORMAT statement is through a character expression containing the format itself. The following sequence of statements is equivalent to the two examples above:

```
CHARACTER*8 FMTCH
FMTCH = '(I5)'
WRITE (*, FMTCH)J
```

The syntax for the format specification statement is:

label FORMAT (field descriptor, field descriptor, ...field descriptor)

A left parenthesis follows the FORMAT statement. The statement ends with a matching right parenthesis. Blank characters can precede the left parenthesis. The processor ignores characters beyond the matching right parenthesis.

Within the parentheses is at least one of the following:

- edit descriptors
- repeat factors
- literal descriptors
- spacing descriptors
- record controls

Table 8-1 gives the format of all repeatable and nonrepeatable edit descriptors available in PC FORTRAN.

Table 8-1. Edit Descriptors

Repeatable	Nonrepeatable
I width	'xxx' (character constants)
G width.decimal places	number H xxx (character constants)
G width.decimal E exponent	number of spaces X (positional editing)
F width.decimal	/ (terminate record)
E width.decimal	(don't terminate record)
E width.decimal E exponent	number P (scale factor)
D width.decimal	BN (blanks as blanks
L width	BZ (blanks as zeros)
A [width]	

8.2 INTERACTION BETWEEN FORMAT AND INPUT/OUTPUT LIST

During execution of the input/output statement, each item in the input/output list is associated with a repeatable edit descriptor. If there is one item in the input/output list, you must use at least one edit descriptor in the format specification statement. You can use an empty edit specification ( ) only if you don't specify any items in the input/output list. When you specify no items in the input/output list, the pointer, the memory location that keeps track of the current item being processed, advances one record. The effect of the empty edit specification upon input, then, is to leave one record out of processing.

Aside from repeatable edit descriptors, no other format control items become associated with an input/output list item; other format control items interact directly with the record.

Repeatable edit descriptors are repeated the number of times indicated by the repeat factor. If you omit the repeat factor, the default is one. A nested format specification is repeated the number of times indicated by the repeat factor. If the rescanning of the format specification begins with a repeated nested format specification, the repeat factor indicates the number of times to repeat the nested format specification. Rescanning does not change the previously set scale factor or the BN or BZ blank control.

The formatted I/O process proceeds as follows. The "format controller" scans the format from left to right. When a repeatable edit descriptor is encountered:

- It is matched with a corresponding item in the input/output list, and I/O of that item proceeds under the format control of the edit descriptor.
- The "format controller" encounters the final parenthesis of the format specification statement and terminates I/O.
- The "format controller" does not find an item corresponding to the repeatable edit descriptor and terminates I/O.
- The "format controller" encounters a right parenthesis, but finds further items in the input/output list. The file is positioned at the beginning of the next record and the "format controller" rescans the format statement at the beginning of the format specification terminated by the last right parenthesis.
- If there is no right parentheses, the "format controller" rescans the format from the beginning. The rescanned format must contain at least one repeatable edit descriptor.
- When the "format controller" terminates upon input, the remaining characters are skipped.
- When the "format controller" terminates upon output, an end of record is written, except when the backslash edit descriptor is used.

8.3 DATA DESCRIPTORS

The data descriptors are I, F, E, G, D, L, and A. Each has a distinctive usage as illustrated in Table 8-2.

Table 8-2. Data Descriptors

Descriptor	Syntax	Usage
I	I width	To format integers Example: FORMAT (I5)
F	F width.decimal	To format real numbers Example: FORMAT (F7.3)
E	E width.decimal E [exponent]	To format real double precision numbers in exponential notation Example: FORMAT (E15.7E-3)
D	D width. significant numbers	To format double precision real numbers Example FORMAT (D13.6)
G	G width.significant numbers	To format numbers of unknown magnitude Example: FORMAT (G13.5)
L	L width	To format logical expressions Example: FORMAT (L5)
A	A [width]	To format alphanumeric expressions Example: FORMAT (A)

8.3.1 The I Format

The I data descriptor describes an integer. No digits other than blanks, interpreted as zeros, digits, or signs are legal. The syntax of the integer editing is:

I width

Width indicates the number of character places in the field.

If the integer output value does not use up all of the spaces indicated in the width, the integer is right-justified and blanks occur to the left of the integer. On input, an optional sign may appear in the field.

8.3.2 The F Format

Real editing formats the input and output for real constants. The constant can be preceded by a plus or a minus sign and contain a decimal point. The syntax of real editing is:

F width.decimal places

The F symbol for real numbers is followed by an integer that indicates the width of the total field, a decimal point and an integer to indicate the number of decimal places within the field. The input field begins with an optional sign followed by a string of digits which may contain an decimal point. If the decimal point is present in the input field, it overrides the decimal point placement in the edit descriptor. Otherwise, the edit descriptor determines the number of decimal places.

Following the decimal places is an optional exponent which is either:

- + or - followed by an integer
- E followed by the following in this order:

  zero or more blanks
  an optional sign
  an integer

The processor rounds the output value rather than truncating it.

The following example formats a real number. The total field is 7 characters wide. The fractional part occupies the last 3 digit places:

```
FORMAT (F7.3)
```

8.3.3 The E Format

The REAL value in D and E editing is output in scientific notation, rounded to a number of significant digits. You generally use the E format when the magnitude of numbers is large or unknown at the time you write the program. The syntax for the E edit descriptor is

E width. significant decimal digits [E exponent]

If the absolute value of the exponent to be printed is larger than 999, you must use the E notation. The processor converts and rounds the E output values and outputs them in the following order.

- a minus sign (if negative)
- a string of digits
- a decimal point
- a string of decimal digits
- the letter E as an exponent indicator
- a sign for the exponent
- exponential digits with possible leading zeros

The values are right justified in the field and blanks fill the unused portion of the field.

In the following example, the field is 14 spaces wide. The 7 most significant digits are printed after which the processor reverts to scientific notation:

```
    PRINT 7, X
7   FORMAT (' ', E14.7)
```

If the value of X in the above example were 325.1234, the processor would print:

```
b0.3251234Eb03  (b=blank)
```

The form of the output field in both E and D editing depends upon the scale factor in effect. The following examples show the E format, the internal value of the number, and the output format with no scale factor in effect:

Format	Internal Value	Output
E12.5	25.571	bb.25571E+02
E9.2	14243.22	bb.14E+05
E6.1	14377.1	1.E+04

8.3.4 The G Format

You can use G editing when you do not know the size of the number beforehand. The G format can replace the F, E, D, I, or L formats. For input, the rules for the G code are the same as for the code that G replaces. The type of input determines which rules apply.

For output, integer and logical codes print according to their rules. For real values, the decimal places indicate the number of significant digits and whether an exponent is printed. If the G format is printed in the F style, the width is -4. If this is not possible because of the field size, it is printed in the E style. The following chart shows how the number is output when n is the magnitude of the number.

Magniture	Equivalent Conversion
n<0.1	E width.decimal
.1<=n<1	F (width-4) decimal bbbb
1<=n<10	F (width-4).(decimal-1)bbbb
10**decimal-2<=n10decimal-1	F(width-4).1bbbb
10decimal-1=n<10decimal	F(width-4).0bbbb
n<10**decimal	Ewidth.decimal

The following examples compares the output of the G and the F format.

Format	Internal Value	Output
G13.6	0.09876543	bb.987654E-01
F13.6	0.09876543	bbbb.09876543
G13.6	987.654321	bb987.654bbbb
F13.6	987.654321	bbb987.654321

8.3.5 The A Format

The A descriptor defines character editing. The syntax of the A format is

A [width]

If you specify the width of the field after the A descriptor, the field consists of the width you specified. If you do not specify the width of the field, the width of the field is the length of the I/O list item.

Upon input, if the width of the field is as large or larger than the input, the processor left-justifies the item and puts blanks in the remainder of the field. Upon output, if the width of the field is larger than the characters in memory, then the processor right-justifies the characters in the output field and fills in the remainder of the space to the left with blanks.

Upon input, if the field is smaller than the input, the processor uses the rightmost characters in the data field. Upon output, if the field is smaller than the character item, it writes only the leftmost characters.

The following chart illustrates how the processor stores and outputs character data:

Descriptor	Internal	External
A2	UVWZ	UV
A4	UVWZ	UVWZ
A6	UVWZ	bbUVWZ

On output, if the width of the field exceed the characters produced by the I/O list item, leading blanks are provided; if the width of the field is small than the number of characters of the I/O list, the leftmost characters are output.

8.3.6 The L Format

The L format is for data type logical. The field consists of two values: T or F. Optional characters can follow these values.

If the value of an item in an output list is 0, the processor outputs F. Otherwise, the processor outputs T. If the width of the field is greater than 1, the processor fills in the remainder of the field with leading blanks. The syntax of the L descriptor is as follows:

L width

The following chart shows the format descriptor, the internal value, and the resulting output:

Format	Internal Value	Output
L1	< >0	T
L3	< >0	bbT
L6	=0	bbbbbF

8.3.7 Blank Interpretation

The BN and BZ edit descriptors tell the processor how to interpret blanks within the numeric input fields. BZ causes the processor to interpret a blank within the input field as a zero. BZ is the default, and it is set at the start of each I/O statement.

BN causes the processor to ignore the blanks within the input field. Ignoring blanks causes the nonblank characters to be right-justified in the field with the leading blanks equal to the number of ignored blanks that originally occurred within the field. The BN stays in effect until the processor encounters a BZ. In the following example, the READ statement accepts the characters shown between the slashes as the value 123.

```
    READ(*, 100) I
100 FORMAT (BN,I6)
    /123       RETURN /
    /123      456 RETURN/
    /      123   RETURN/
```

Reading short records can temporarily invoke the BN status automatically. If the total number of characters in the input record is fewe than those specified by the combination of format descriptors and the I/O lis elements, the record is padded on the right with blanks to the required length, and BN editing goes into effect temporary. Thus the following exampl results in the value 123 rather than 12300.

```
    READ (*, '(I5)' I
    /123 RETURN/
```

8.4 REPEAT FACTORS

At times, you may wish to repeat the same descriptor. Repeat specifications allow you to do this. The repeat specification occupies the position immediately before the data descriptor or group and tells how many times to repeat the same data descriptor. It is composed of an unsigned nonzero integer constant. The following statement causes 10 variables in the I/O lis to be output according to the format F6.3.

```
    210 FORMAT (10F6.3)
```

A group, a parenthesized sublist of FORMAT descriptors, allows you to repeat many descriptors with one command. You can nest format specifications up to three levels within the outermost level.

The following FORMAT statement causes A5 to repeat twice and 1X and 3A8 to repeat 3 times. A8 is, therefore, repeated 9 times in all.

```
    610 FORMAT (2A5, 3(1X, 3A8)) A1, CHAT
```

8.5 NONREPEATABLE EDIT DESCIPTORS

The nonrepeatable edit descriptions are described in the following subsection

3.5.1 Apostrophe editing

The processor interprets character strings literally. There are two ways of writing character strings in FORTRAN: through the use of single quotations and through the use of the Hollerith fields.

In the following example, the processor writes the literal string exactly. Embedded blanks are significant. Since single apostrophes surround the character constant, two consecutive apostrophes ('') represent a single apostrophe within the string.

```
10 FORMAT ('     This is a message!')
```

The syntax of this type of character editing is:

'character constants of any length'

3.5.2 Hollerith Editing

A second type of literal character editing is called Hollerith editing. In Hollerith editing, the count for the number of characters in the field precedes an H which is followed by the field of characters. In Hollerith editing, blanks are significant. Hollerith editing cannot be used for input (READ).

The format of Hollerith editing is:

total number of characters in field H ASCII string

The following figure shows the input and output of Hollerith fields.

Format	Input	Output
4HJOHN	GREG	GREG
7HbbHALVE	HALFbbb	HALFbbb

In the following example, "This is a message" is composed of 17 characters:

```
FORMAT ('1', 17HThis is a message)
```

8.5.3 Positional Editing

On input (READ), the spacing descriptor causes the file position to advance n characters, thus skipping n characters. On output (WRITE), nX causes n blanks to be written. The syntax of the spacing descriptor is:

nX

The following FORMAT statement causes a line on which "bread" and "cheese" is separated by ten spaces:

```
      WRITE (6,1000)
10    FORMAT('bread', 10X, 'cheese')
```

8.5.4 Slash Editing

The slash indicates the end of data transfer on the current record. On input, the file is positioned to the beginning of the next record. On output, an end-of-record is written, and the file is positioned to write on the beginning of the next record. Through the use of the slash, one FORMAT statement can describe more than one record. In the following example, the READ statement reads three separate records: two values from the first record, two from the second record, and one value from the third record:

```
      READ (5, 10) Q,R,S,T,U
10    FORMAT(2F5.2/ 2F6.3/ F2.1)
```

8.5.5 Backslash Editing

Normally when the format controller terminates, the end of data transmission on the current record occurs. If the last edit descriptor encountered is the backslash, this automatic end-of-record is inhibited, allowing for subsequent I/O statements to continue reading or writing out of or into the same record.

In the following statement, a prompt appears to which the user responds. The backslash in the WRITE statement inhibits the end of the record so that the user's response can provide input to the READ statement:

```
WRITE (*, '(A )') 'Input an integer'
READ (*,('BN, I6)')I
```

The backslash does not inhibit the automatic end of record generated when reading from the keyboard, thus allowing the backspace character and the DELETE key to function properly. Input from the keyboard must be terminated with the RETURN key.

8.5.6 Scale Factor Editing

The scale factor,used only E and F edit discriptors, allows you to shift the decimal point to the left or right during an input or output operation. At the start of each I/O statement, the scale factor is initialized to zero. The general format of the scale factor is:

nP

N is a signed or unsigned integer.

The scale factor is used as a prefix for either an E or F format specification and stays in effect until it the processor encounters another scale factor edit descriptor or the input and output terminates. You can disable the scale factor by indicating a scale factor of 0.

The scale factor causes the value of the input or output to be multiplied or divided by the power of ten specified in the statement. The following table summarizes the operation of the scale factor during an I/O operation:

Format	Effect on input	Effect on Output
F	Multiplies by 10**-n	Multiplies by 10**n
E	Multiplies by 10**-n Increases exponent by n	Multiplies by 10**n Reduces exponent by n

If there is already an explicit exponent in the input field, the scale factor has no effect.

The following examples show the effect of the scale factor on output:

Format	Input	Internal Value	Output
2PF8.2	21	.0021 (.21*10-2)	.21
-2PF5.2	56.63	.5663	56.63
2PE8.2	56.63E2	5663.	56.63E2
0PF5.1	21.2	21.2	21.2

8.6 CARRIAGE CONTROL

A carriage control character, if present, is the first item of the FORMAT statement. Whether or not it is present, the processor interprets the first character of every record transferred to a printer as a carriage control character. These characters and their effects are listed below.

CHARACTER	EFFECT
space	advances one line
0	advances two lines
1	advances to top of next page
+ (plus)	does not advance (allows overprinting)

The following FORMAT statements illustrate the use of the carriage control characters. In the first, the printer advances one line; in the second, the printer advances two lines; in the third, the printer advances to the top of the next page; and in the last, the printer does not advance at all to allow for overprinting:

```
FORMAT (' ' I5)
FORMAT ('0' I5)
FORMAT ('1' I5)
FORMAT ('+' I5)
```

The processor treats any other character in the first character space of a record as a space and deletes it from the print line. If you accidentally omit the carriage control character, the processor does not print the first character of the record.

9

Programs, Subroutines and Functions

CHAPTER 9
PROGRAMS, SUBROUTINES, AND FUNCTIONS

9.1 INTRODUCTION

A FORTRAN program consists of one main program and an optional number of subprograms and functions. Subroutines and functions let you develop large structured programs that can be broken into parts. This is advantageous in the following situations:

- If a program is large, breaking it into parts makes it easier to develop, test, and maintain.
- If a program is large and recompiling the entire source file is time consuming, breaking the program into parts saves compilation time.
- If you intend to include certain routines in a number of different programs, you can create a single object file that contains these routines and then link it to each of the programs in which the routines are used.
- If a routine could be implemented in any of several ways, you might place it in a file and compile it separately. Then, to improve performance, you can alter the implementation, or even rewrite the routine in assembly language or in Pascal, and the rest of your program will not need to change.

9.2 THE MAIN PROGRAM

The main program is not subordinate to another program. The first statement in a main program is not a FUNCTION or SUBROUTINE statement, but can be a PROGRAM statement. If there is no PROGRAM statement, the main program is assigned the default name MAIN. There must be one main program in every executable program.

The execution of a program always begins with the first executable statement in the main program.

A main program can have a PROGRAM statement as its first statement which identifies the program unit as a main program and gives it a name. If a PROGRAM statement is present, it must be the first statement in the main program. The syntax of a PROGRAM statement is:

PROGRAM program name

The program name is a user-defined global name. Therefore, it cannot have the same name as that of another external procedure or common block. Since it is a local name to the main program, it must not conflict with any other local name in the main program. If you use the default name MAIN, you cannot use it to name any other entity in the program..

In the following example, the PROGRAM statement names the program MYPROG:

```
PROGRAM MYPROG
    .
    .
    .
END
```

3.3 SUBROUTINES

A subroutine is a subordinate program unit that can be called from other program units. Subroutines help you to organize your program into small, logical parts. When the subroutine is invoked, it performs the set of actions defined by its executable statements. Values can be passed back to the calling program unit via arguments or common variables in a subroutine.

A subroutine begins with a SUBROUTINE statement and ends with an END statement. The SUBROUTINE statement identifies a program unit as a subroutine, gives it a name, and identifies the formal arguments to that subroutine. A subroutine can contain any kind of statement other than a PROGRAM statement, another SUBROUTINE statement, or a FUNCTION statement. The syntax of a SUBROUTINE statement is:

```
SUBROUTINE subroutine name ([[formal argument [,formal argument ]...])]
    .
    .
RETURN
END
```

The list of argument names defines the number and, along with any subsequent IMPLICIT, EXTERNAL, type, or DIMENSION statements, the type of arguments to that subroutine. Argument names cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

The actual arguments in the CALL statement that reference a subroutine must agree with the corresponding formal arguments in the SUBROUTINE statement in order, number and type. The SUBROUTINE statement must precede the CALL statement in the current compilation for the compiler to check the correspondence between the actual arguments and the formal arguments.

The following subroutine prints the squares of the numbers from 1 to 1

```
    SUBROUTINE SQUARES (J)
    INTEGER N, J
    N=1
10  J=N*N
    WRITE (*,20) N, J
20  FORMAT ("THE SQUARE OF" I3 "=" I3)
    N=N+1
    If (N.LE.10) GOTO 10
    RETURN
    END
```

9.3.1 Arguments

Data is transmitted to and from a subroutine or function subprogram through the use of arguments. A formal argument is the name by which the argument is known within a function or subroutine. An actual argument is the specific variable, expression, array, etc. passed to the procedure at any specific calling location.

The first statement in a subroutine or function, called a header, usually lists formal arguments that must agree in type and number with the corresponding arguments of the CALL statement.

Upon entry to a subroutine or function, the actual arguments are associated with the formal arguments. This association remains in effect until the execution of the subroutine or function is terminated. If an actual argument is an expression, it is evaluated immediately prior to the association of formal and actual arguments. If an actual argument is an array element, its subscript expressions are evaluated just prior to the association and remain constant through the execution, even if they contain variables that are redefined during the execution of the procedure.

The following rules apply to arguments:

- The list of arguments in a subroutine is optional. You can omit them if the subroutine accepts or returns no outside information.
- Arguments can include the names of:
  - external procedures
  - functions
  - intrinsic functions
  - arrays
  - variables
  - constants
  - expressions

- A formal argument that is a variable can be associated with an actual argument that is a variable, an array element, or an expression.

- If an actual argument is a constant, function reference, or an expression other than a simple variable, assigning a value to the corresponding formal argument is not permitted.

- Assigning a value to a formal argument of type CHARACTER when the actual argument is a literal can produce unpredictable results

- If the argument is a procedure or function, it must appear in an EXTERNAL or INTRINSIC statement in the program unit in which the reference occurs.

- Except for the number of dimensions of an array, the number and type of arguments of the calling statement and of the subprogram must agree. If a CALL statement passes two arguments, an integer variable and a real array in that order, the arguments of the subroutine must be an integer variable followed by a real array.

- A formal argument that is an array can be associated with an actual argument that is an array or an array element. The number and size of dimensions in a formal argument may be different from those of the actual argument, but any reference to the formal array must be within the limits of the memory sequence in the actual array. The compiler does not flag an error when a reference to an array is outside the bounds of memory, but the results are unpredictable.

- You should avoid assigning a value to a formal argument during execution. Doing so may alter the value of the corresponding actual argument.

- The names of the arguments cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

- A formal argument can be associated with an external subroutine, function, or intrinsic function if it is used in the body of the procedure as a subroutine or function reference, or if it appears in an EXTERNAL statement. The corresponding actual argument must be an external subroutine or function, declared with the EXTERNAL statement, or an intrinsic function permitted to be associated with a formal procedure argument. The intrinsic function must have been declared with an INTRINSIC statement in the program unit where it is used as an actual argument.

- All intrinsic functions except the following can be associated with formal procedure arguments.

INT	IFIX	IDINT	FLOAT	SNGL	REAL	DBLE
ICHAR	CHAR	LGE	LGT	LLE	LLT	MAXO
AMAX1	DMAX1	AMAXO	MAX1	MINO	AMIN1	DMIN1
AMINO	MIN1					

9.3.2 The CALL Statement

A CALL statement transfers control to a subroutine. When the processor encounters a CALL statement it does the following:

- evaluates all arguments that are expressions
- associates all arguments with their corresponding formal arguments
- executes the body of the subroutine
- returns control to the statement following the CALL statement by executing either a RETURN or an END statement in that subroutine

The syntax of a CALL statement is:

CALL subroutine name [([argument [,argument]...])]

The following rules apply to the CALL statement:

- The argument can be
  - an expression
  - a variable
  - a constant (or constant expression)
  - an array element
  - a subroutine
  - an array
  - an external function
  - an intrinsic function permitted to be passed as an argument
- The arguments in the CALL statement must agree in order, type, and number with the corresponding arguments in the SUBROUTINE statement. If there are no arguments in the SUBROUTINE statement, then a CALL statement referencing that subroutine has no arguments. An optional pair of parentheses can follow the name of the subroutine, however.
- The compiler checks for correspondence between the formal argument and the arguments. For the formal argument to be known, the SUBROUTINE statement that defines the formal arguments must precede the CALL statement.

- You can use a formal argument as an argument in a CALL statement for another subprogram.

- You can call a subroutine from any program unit.

- Except through the use of the $DYNAMIC metacommands, recursive subroutine calls are not permitted in FORTRAN. That is, a subroutine cannot call itself directly and it cannot call upon another subroutine that results in that subroutine being called again before it returns control to its caller.

If the arguments are integer or logical values, agreement in size is required according to the following rules.

- If the formal argument is unknown, its size is determined by the $STORAGE metacommand (except as noted in rule % of this list). If the $STORAGE metacommand is not specified, the default is $STORAGE:4.

- If the actual argument is a constant (or constant expression), and the size of the actual argument is smaller than the size of the formal argument, a temporary variable the size of the constant is created for the actual argument. If the actual argument is larger, an error is generated.

  95 argument type conflict

- If the actual argument is an expression smaller than the size of the formal argument, the size of the formal argument is created. This is a temporary variable. If the actual argument is larger, error code 95 is generated (refer to the preceding rule).

- If the actual argument is an array element and the formal argument an array, or if the actual argument is an array or a function, the compiler does not check for agreement in size.

- If the actual argument is a variable or an array element and the formal argument is unknown, the size of the formal argument is assumed to be the same size as the size of the actual argument. This allows you to call separately compiled subroutines whose formal argument differ from the size determined by the $STORAGE metacommand in effect. If a known formal argument is larger than the actual argument, a temporary variable for the actual argument is created. If the formal argument is unknown, error code 95 is generated as in previous examples.

In the following example, if the argument IERR does not equal zero, subroutine ERROR sends out an error statement:

```
      IF (IERR .NE. 0) CALL ERROR(IERR)
      END

      SUBROUTINE ERROR(IERRNO)
      WRITE (*, 200) IERRNO
  200 FORMAT(1X, 'ERROR', I5, 'DETECTED')
      END
```

9.3.3 The RETURN Statement

The RETURN statement causes the processor to return control to the calling program unit. This statement can only appear in a function or a subroutine.

The syntax of the RETURN statement is:

```
RETURN
```

When the processor encounters a RETURN statement, it terminates the execution of the subroutine or function. If the RETURN statement is in a function, the value of the function is equal to the variable with the same name as that of the function.

Execution of an END statement in a function or subroutine is equivalent to the execution of a RETURN statement. Thus, either a RETURN or an END statement, but not both, is required to terminate a function or subroutine.

In the following example, the subroutine loops until you type in Y.

```
      SUBROUTINE LOOP
      CHARACTER IN
   10 READ(*,'A1)')  IN
      IF (IN .EQ. 'Y') RETURN
      GOTO 10
      RETURN
      END
```

9.4 FUNCTIONS

A function returns a single value. A function reference may appear in an arithmetic or logical expression. Upon encountering the reference, the processor evaluates the function and uses the resulting value as an operand in the expression.

There are three kinds of functions: subprograms, statement, and intrinsic.

The format of a function reference is as follows.

function name ([argument [, argument] ...])

The rules for arguments for functions are identical to those for subroutines. Refer to Subsection 9.2.2 for more details.

9.4.1 Function Subprograms

A function subprogram is very similar to an ordinary subprogram. The major difference between subprograms and functions is the way that they transmit values back to the calling program. The subprogram transmits values back through the arguments of the subroutine while the function transmits the value back through its FUNCTION name. The function name acts like an ordinary variable.

The FUNCTION statement identifies a program unit as a function and supplies its type, name, and optional formal arguments. The function subprogram begins with a FUNCTION statement and ends with an END statement. It can contain any kind of statement except a PROGRAM, FUNCTION, or a SUBROUTINE statement.

The syntax of a FUNCTION statement is:

```
[type] FUNCTION function name  ([argument [,argument] ...])
          .
          .
          .
       RETURN
       END
```

The following rules apply to function subprograms:

- The name of the arguments with their accompanying IMPLICIT, type, or DIMENSION statements define the number and the type of arguments to that subroutine.
- If you do not specify the type in the FUNCTION statement, the default or any subsequent IMPLICIT or EXTERNAL type statements determine the type.
- If a type is present, the function name cannot appear in any additional type statements.
- The type in the FUNCTION statement is INTEGER (*2, *4), REAL (*4, *8), DOUBLE PRECISION, or LOGICAL (*2, *4).

- An external function cannot be of type CHARACTER.
- Argument names and function names cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.
- The function name, a user-defined name, is a global name.
- The function name must appear as a variable in the program unit defining the function. Every execution of that function must assign a value to the variable. Upon the RETURN or END statement, the final value defines the value of the function.
- After a function is defined, you can reference the value exactly as you can any other variable.
- An external function can return values in addition to the value of the function by assignment to one or more of its formal arguments.
- You can call a function from any program unit.
- Recursive function calls are illegal. That is, a function cannot call itself directly, nor can it call another function that calls upon itself.

The following function reads a number from a file and returns that number to the main program unit:

```
    I=2
10  IF (GETNO(I).EQ. 0.0) GO TO 10
    STOP
    END

    FUNCTION GETNO(NOUNIT)
    READ(NOUNIT, '(F10.5)') R
    GETNO = R
    RETURN
    END
```

9.4.2 Statement Functions

A statement function definition is a nonexecutable single arithmetic or logical assignment statement. The body of a statement function defines the meaning of the statement function. It is executed when the expression in which it appears is executed.

The syntax of the statement function is:

function name ([argument [argument]...]) = expression

You call a statement function by its name. Following the name are a parenthesized list of arguments. The processor evaluates the expression by using these arguments and assigns the results to the function name.

The following rules apply to statement functions:

- The expression following the function name can contain references t
  - dummy arguments
  - non-literal constants
  - variables
  - array elements
  - utility and mathematical functions
  - previously defined statement functions
- A statement function name can only appear after the specification statements and before any executable statements in the program unit in which it appears.
- You cannot call statement functions recursively, either directly or indirectly.
- The list of formal arguments defines the number and type of arguments. The corresponding argument in every argument list must agree in type with the corresponding argument in the statement function definition.
- The statement function can be of type real, integer, or logical, but not of type character.
- You can use the type statement to define the type of the statement function.
- The name of a statement function can appear in a type statement and in a COMMON statement. It cannot appear in any other specification statements.
- The type statement cannot define the function statement as an array name.
- As with other variables, the first character of the function name determines the default type.
- The statement function can only appear after the specification statements and before any executable statements.
- If a formal argument name is the same as another local name, a reference to that name within the statement function refers to the formal argument name.
- Since the name of the statement function is local to the enclosing program unit, you can only reference the statement function in the program unit in which you define it. The exceptions to this rule is that you can reference the statement function outside the program unit as the name of a common block or as a formal argument to another statement function

The following example illustrates a statement function definition and a call in which FUNCT1 is subtracted from A:

The definition: FUNCT1(E,F,G)=((E*F)/G)

The function call: A=A-FUNCT1(T,U,V)

9.4.3 Intrinsic Functions

The FORTRAN compiler predefines intrinsic functions and makes them available for use in a FORTRAN program. To use the intrinsic functions in your program, write their names followed by the appropriate arguments.

The syntax of the intrinsic function is:

function name ([argument [argument]...])

The arguments of the function must agree in type, number, and order with the specifications in Table 8. If the function demands a real constant, the expression must evaluate to be a real constant, for example. If an intrinsic function allows several types of arguments, all arguments in a single reference must be of the same type.

For example, the following intrinsic function converts I7 to real and assigns the result to the variable VAR:

```
VAR=FLOAT(I7)
```

The processor treats the function as if it were a variable. You can, therefore, use the function as a component of an expression.

You can not alter the type of an intrinsic function. You can use the intrinsic function name in a type or INTRINSIC statement, but you must define it as the same type as that of the intrinsic function.

The definition of the function limits the arguments. For example, the logarithm of a negative number is mathematically undefined, and therefore not permitted.

Table 8 gives the name, definition, number of arguments, and type of the intrinsic functions available in Wang Professional Computer FORTRAN.

In Table 9-1, all angles are expressed in radians.

Table 9-1. Intrinsic Functions

Name	Definition	Type of Argument	Type of Function
TYPE CONVERSION			
INT(X) [1]	Conversion to integer	Real*4	Integer
		Integer	Integer
IFIX(X)	Conversion to integer	Real*4	Integer
IDINT (2)	Conversion to integer	Real*8	Integer
REAL(X) [2]	Conversion to real*4	Integer	Real*4
FLOAT(I)	Conversion to real*4	Integer	Real*4
ICHAR(C) [3]	Conversion to integer	Character	Integer
CHAR(X) [3]	Conversion to character	Integer	Character
SNGL(X)	Conversion to real*4	Real*8	Real*4
DBLE(X) [4]	Conversion to real*8	Integer	Real*8
		Real*4	Real*8
		Real*8	Real*8
TRUNCATION			
AINT(X)	Truncation to real*4	Real*4	Real*4
DINT(X)	Truncation to real*8	Real*8	Real*8
NEAREST WHOLE NUMBER			
ANINT(X)	Truncate to real*4	Real*4	Real*4
DINT(X)	Truncate to real*8	Real*8	Real*8
NEAREST INTEGER			
NINT(X)	Rounding to integer	Real*4	Integer
IDNINT(X)	Rounding to integer	Real*8	Integer
ABSOLUTE VALUE			
IABS(I)	Integer absolute	Integer	Integer
ABS(X)	Real*4 absolute	Real*4	Real*4
DABS(X)	Real*8 absolute	Real*8	Real*8
REMAINDERING			
MOD(I,J)	Integer remainder	Integer	Integer
AMOD(X,Y)	Real*4 remainder	Real*4	Real*4
DMOD(X,Y)	Real*8 remainder	Real*8	Real*8
TRANSFER OF INTEGER SIGN			
ISIGN(I,J)	Integer transfer	Integer	Integer
SIGN(X,Y)	Real*4 transfer	Real*4	Real*4
DSIGN(X,Y)	Real*8 transfer	Real*8	Real*8

continued

Table 9-1. Intrinsic Functions (continued)

Name	Definition	Type of Argument	Type of Function
POSITIVE DIFFERENCE			
IDIM(I,J)	Integer difference	Integer	Integer
DIM(X,Y)	Real*4 difference	Real*4	Real*4
DDIM(X,Y)	Real*8 difference	Real*8	Real*8
CHOOSING LARGEST VALUE			
MAX0(I,J,...)	Integer maximum	Integer	Integer
AMAX1(X,Y,...)	Real*4 maximum	Real*4	Real*4
AMAX0(I,J,...)	Real*4 maximum	Integer	Real*4
MAX1(X,Y,...)	Integer maximum	Real*4	Integer
DMAX1(X,Y,...)	Real*8 maximum	Real*8	Real*8
CHOOSING SMALLEST VALUE			
MIN0(I,J,...)	Integer minimum	Integer	Integer
AMIN1(X,Y,...)	Real*4 minimum	Real*4	Real*4
AMIN0(I,J,...)	Real*4 minimum	Integer	Real*4
MIN1(X,Y,...)	Integer minimum	Real*4	Integer
DMIN1(X,Y,...)	Real*8 minimum	Real*8	Real*8
SQUARE ROOT			
SQRT	Square root	Real*4	Real*4
DSQRT	Real*8 square root	Real*8	Real*8
EXPONENTIAL			
EXP(X)	Real*4 e raised to power	Real*4	Real*4
DEXP(X)	Real*8 e raised to power	Real*8	Real*8
NATURAL LOGARITHM			
ALOG(X)	Natural logarithm of Real*4	Real*4	Real*4
DLOG(X)	Natural logarithm of Real*8	Real*8	Real*8
COMMON LOGARITHM			
ALOG10(X)	Common logarithm of Real*4	Real*4	Real*4
DLOG10(X)	Common logarithm of Real*8	Real*8	Real*8
SINE			
SIN(X)	Real*4 sine	Real*4	Real*4
DSIN(X)	Real*8 sine	Real*8	Real*8

Table 9-1. Intrinsic Functions (continued)

Name	Definition	Type of Argument	Type of Function
COSINE			
COSIN(X)	Real*4 cosine	Real*4	Real*4
DCOSIN(X)	Real*8 cosine	Real*8	Real*8
TANGENT			
TAN(X)	Real*4 tangent	Real*4	Real*4
DTAN(X)	Real*8 tangent	Real*8	Real*8
ARC SINE			
ASIN(X)	Real*4 arc sine	Real*4	Real*4
DASIN(X)	Real*8 arc sine	Real*8	Real*8
ARC COSINE			
ACOS(X)	Real*4 arc cosine	Real*4	Real*4
DACOS(X)	Real*8 arc cosine	Real*8	Real*8
ARC TANGENT			
ATAN(X)	Real*4 arc tangent	Real*4	Real*4
DATAN(X)	Real*8 arc tangent	Real*8	Real*8
ATAN2(X/Y)	Real*4 arc tangent of X/Y	Real*4	Real*4
DATAN2(X/Y)	Real*8 arc tangent of X/Y	Real*8	Real*8
HYPERBOLIC SINE			
SINH(X)	Real*4 hyperbolic sine	Real*4	Real*4
DSINH(X)	Real*8 hyperbolic sine	Real*8	Real8*
HYPERBOLIC COSINE			
COSH(X)	Real*4 hyperbolic cosine	Real*4	Real*4
DCOSH(X)	Real*8 hyperbolic cosine	Real*8	Real*8
HYPERBOLIC TANGENT			
TANH(X)	Real*4 hyperbolic tangent	Real*4	Real*4
DTANH(X)	Real*8 hyperbolic tangent	Real*8	Real*8
LEXICALLY GREATER THAN OR EQUAL			
LGE(C1,C2)	First argument greater than or equal to second	Character	Logical

Table 9-1. Intrinsic Functions (continued)

Name	Definition	Type of Argument	Type of Function
LEXICALLY GREATER THAN LGT(C1,C2)	First argument greater than second	Character	Logical
LEXICALLY LESS THAN OR EQUAL LLE(C1,C2)	First argument less than or equal to second	Character	Logical
LEXICALLY LESS THAN LLT(C1,C2)	First argument less than second	Character	Logical
END OF FILE EOF(X)	Integer end of file	Integer	Logical

NOTES:
For X of type INTEGER, INT(X)=X. For X of type REAL or REAL*8, if X is greater than or equal to zero, INT(X) is the largest integer not greater than X, and if X is less than zero, INT(X) is the most negative integer not less than X. For X of type REAL, IFIX(X) is the same as INT(X).

For X of type REAL, REAL(X)=X. For X of type INTEGER or REAL*8, REAL(X) is as much precision of the significant part of X as a real datum can contain. For X of type INTEGER, FLOAT(X) is the same as REAL(X).

For X of type REAL*8, DBLE(X)=X. For X of type INTEGER or REAL, DBLE(X) is as much precision of the significant part of X as a double precision datum can contain.

INT(X) and IFIX(X) truncate the fractional part of the real number.

LGE(X,Y) returns the value .TRUE. if X = Y or if X follows Y in the ASCII collating sequence; otherwise it returns .FALSE..

LGT(X,Y) returns .TRUE. if X follows Y in the ASCII collating sequence; otherwise it returns .FALSE..

LLE(X,Y) returns .TRUE. if X = Y or if X precedes Y in the ASCII collating sequence; otherwise it returns .FALSE..

LLT(X,Y) returns .TRUE. if X precedes Y in the ASCII collating sequence; otherwise it returns .FALSE.. If the operands are of unequal length, the shorter operand is considered to be blankfilled on the right to the length of the longer operand.

The operands of LGE, LGT, LLE, and LLT must be of the same length.

EOF(a) returns the value .TRUE. if the unit specified by its argument is at or past the end of file record; otherwise it returns .FALSE.. The value of A must correspond to an open file or to unit zero (the screen or keyboard device).

ICHAR converts a character value into the integer value worth the ASCII internal representation of that character. This value is within the range 0 to 255. One character is worth less than another only if it is worth less when used with ICHAR as follows.

```
(ICHAR(C1) .LE. ICHAR(C2))
```

CHAR n returns the nth character in the collating sequence. The value is of type CHARACTER, length one, while n must be an integer expression whose value is in the range of 0=n=255.

ICHAR (CHARn)= n for 0=n=255.

CHAR(ICHAR(c)) = c for any character c in the character set.

10

Metacommands

CHAPTER 10
METACOMMANDS

10.1 INTRODUCTION

Metacommands are compiler directives that order the FORTRAN compiler to process source text in particular ways. You can intermix metacommands with text within a source program; however, they are not part of the FORTRAN language.

The compiler interprets a line of input that begins with a dollar sign ($) to be a metacommand. A metacommand and its arguments must fit on a single source line. Continuation lines and blanks within the metacommand line are not legal.

There are three kinds of metacommands in Wang Professional Computer FORTRAN: those that help with debugging; those that are programming aids; and those that help with formatting. Table 10-1 lists the metacommands and their functions.

Table 10-1. Metacommands

Metacommand	Function
$DEBUG	Turns on runtime checking for arithmetic operations and assigned GOTO. $NODEBUG turns checking off.
$DO66	Causes DO statements to have FORTRAN 66 semantics.
$INCLUDE	Directs the compiler to proceed as if the indicated file were inserted at this point.
$LINESIZE	Makes subsequent pages of the listing file the number of columns wide that is indicated.
$LIST	Sends subsequent listing information to the listing file. $NOLIST stops generation of listing information.
$PAGE	Starts new page of listing.

continued

Table 10-1. Metacommands (continued)

Metacommand	Function
$PAGESIZE	Makes subsequent pages of listing file the number of lines long that is indicated.
$STORAGE	Allocates the indicated number of bytes of memory to all LOGICAL or INTEGER variables in the source file.
$STRICT	Disables PC FORTRAN features not in the 1977 subset or full language standard. $NOTSTRICT enables them.
$TITLE	Gives a title for the subsequent pages of the listing.

10.2 THE DEBUGGING METACOMMANDS

A number of metacommands help you to debug your program. The DEBUG and NODEBUG metacommands allow you to test for certain errors while the LIST and NOLIST metacommands give you some control of the listing file.

10.2.1 THE DEBUG AND NODEBUG METACOMMANDS

The DEBUG Metacommand tests for four possible errors in the your program:

- overflow in an arithmetic operation
- division by zero
- invalid index values
- invalid values in an assigned GOTO statement

In addition, it provides the runtime error-handling system with source file names and line numbers. If any of the above errors occur in your program, a runtime error is generated, and the source line and file name are displayed on the screen.

The NODEBUG metacommand turns off the DEBUG metacommand. It is the default value.

The syntax of DEBUG/NODEBUG is:

```
$DEBUG
$NODEBUG
```

The DEBUG/NODEBUG metacommands can appear anywhere in your program.

10.2.2 The LIST/NOLIST Metacommands

The LIST metacommand directs the compiler to list subsequent commands in the listing file. Although LIST is the default condition, the metacommand has no effect if you do not specify a listing file when you invoke the compiler.

NOLIST directs the compiler to refrain from sending subsequent listing information to the listing file.

The format for the LIST/NOLIST metacommand is:

```
$LIST
$NOLIST
```

You can place $LIST/$NOLIST anywhere in a source file.

10.3 METACOMMANDS THAT AID PROGRAMMING

A number of metacommands help you to control the data, input, or commands in your program. They are: DO66, STRICT/NOSTRICT, INCLUDE and STORAGE.

10.3.1 The DO66 Metacommand

Through the DO66 metacommand, DO statements have FORTRAN 66 rather than FORTRAN 77 semantics. FORTRAN 66 semantics differ from FORTRAN 77 ANSI standard semantics in a number of ways:

- The processor executes DO statements at least once in FORTRAN 66; if the initial value has a negative increment or is greater than the final control variable in FORTRAN 77, the processor does not execute the DO statement at all.
- In FORTRAN 66, control may transfer in and out of the syntactic body of the DO statement; therefore, the range of the DO statement includes any statement that the processor can execute between the DO statement and its terminal statement. In FORTRAN 77, this extended range is illegal.

The syntax for the DO66 metacommand is:

```
$DO66
```

10.3.2 The STRICT/NOTSTRICT Metacommands

The STRICT metacommand disables the specific PC FORTRAN features not found in the ANSI FORTRAN 77 subset or full language standard. STRICT is the default.

The default, NOTSTRICT metacommand, has the opposite effect. It allows you to use PC FORTRAN features not found in the ANSI FORTRAN 77 subset or full language standard. These features are as follows:

- Character expressions may be assigned to noncharacter variables.
- Character and noncharacter expressions can be compared.
- Character and noncharacter variables are allowed in the same COMMON block.
- Character and noncharacter variables can be made equivalent.
- Noncharacter variables can be initialized with character data.

The format for the STRICT/NOTSTRICT metacommand is:

```
$STRICT
$NOTSTRICT
```

$STRICT/$NOTSTRICT can appear anywhere in a source file.

10.3.3 The INCLUDE Metacommand

The INCLUDE metacommand directs the compiler to proceed as though the specified file were inserted at the point of the $INCLUDE. After including the outside file, the compiler resumes processing the original source file on the line following INCLUDE.

The INCLUDE metacommand is particularly useful in guaranteeing that several modules use the same declaration for a COMMON block.

The format for the INCLUDE metacommand is:

$INCLUDE: name of file

The compiler imposes no limit on nesting levels for INCLUDE metacommands.

10.3.4 The STORAGE Metacommand

The STORAGE metacommand allocates either 2 or 4 bytes of memory for all variables declared in the source file as INTEGER or LOGICAL.

STORAGE does not affect the allocation of memory for variables which you have declared with an explicit length specification. For example, INTEGER*2 or LOGICAL*4 already have specific lengths.

When you link several files of a source program together, the allocatic of memory for variables common to the program units, such as arguments and parameters, must be consistent.

The format for the STORAGE metacommand is:

$STORAGE: number of bytes

The number of bytes can be either 2 or 4.

If a program contains no STORAGE metacommand, the compiler uses the default of 4 for INTEGER, LOGICAL, and REAL variables. Setting the STORAGE metacommand to 2 allows source programs to run faster and use less code.

The STORAGE metacommand must precede the first declaration statement of the source file in which it occurs.

10.4 THE FORMATTING METACOMMANDS

PC FORTRAN includes five metacommands that allow you to format the program page: PAGE, TITLE, SUBTITLE, LINESIZE, and PAGESIZE.

10.4.1 The PAGE Metacommand

The PAGE metacommand directs the processor to begin a new page.

The format for the PAGE metacommand is:

$PAGE

If the first character of a line of source test is the ASCII form feed character it is considered equivalent to the occurrence of a PAGE metacommand at that point.

10.4.2 The TITLE Metacommand

The TITLE metacommand directs the compiler to include a specific title on the following pages of the listing. A new TITLE metacommand overrides the previous one.

The format for the TITLE metacommand is:

$TITLE: 'title'

The title is any valid character constant up to 40 characters long. If a program contains no TITLE metacommand, the default is a null string.

10.4.3 The SUBTITLE Metacommand

The SUBTITLE metacommand directs the processor to include a subtitle in the following pages of the source listing. This subtitle is overridden by a new one the next time the processor encounters a $SUBTITLE metacommand.

The format for the SUBTITLE metacommand is:

$SUBTITLE: 'a subtitle'

The subtitle is any valid character constant up to 40 characters long.

10.4.4 The LINESIZE Metacommand

The LINESIZE metacommand directs the processor to create a line a certain number of characters wide.

The format for the LINESIZE metacommand is:

$LINESIZE: number of characters

The number of characters has a maximum value of 132 and a minimum value of 40. The default size is 80.

10.4.5 The PAGESIZE Metacommand

The PAGESIZE metacommand directs the processor to create a page a certain number of lines long. The format for the PAGESIZE metacommand is:

$PAGESIZE: number of lines

The number of lines is any positive integer greater than 15. If a program contains no PAGESIZE metacommand, the default is 66.

2200-2249 Type INTEGER*4 Arithmetic Errors
2200 Long integer divided by zero
2201 Long integer math overflow
2234 Long integer zero to negative power

Other Errors

2451 Assigned GOTO label not in list

APPENDIX A
ERROR MESSAGES

A.1 INTRODUCTION

This appendix lists error messages and the codes for errors detected by the compiler and runtime systems.

A.2 COMPILETIME ERROR MESSAGES

Code	Message
1	fatal error reading source
2	nonnumeric characters in label field
3	too many continuation lines
4	fatal end of file encountered
5	label in continuation line
6	missing field in metacommand
7	cannot open file
8	unrecognizable metacommand
9	input file invalid format
10	too many nested include files
11	integer constant overflow
12	real constant error
13	too many digits in constant
14	identifier too long
15	character constant not closed
16	zero length character constant
17	invalid character in input
18	integer constant expected
19	label expected
20	label error
21	type expected
22	integer constant expected
23	extra characters at end of statement
24	"(" expected
25	letter already used in IMPLICIT
26	")" expected
27	letter expected
28	identifier expected
29	dimensions expected
30	array already dimensioned
31	too many dimensions
32	incompatible arguments

Error Code	Message
33	identifier already has type
34	identifier already declared
35	INTRINSIC FUNCTION not allowed here
36	identifier must be a variable
37	identifier must be a variable or the current FUNCTION
38	"/" expected
39	named COMMON block already saved
40	variable already appears in COMMON
41	variables in two different COMMON blocks
42	number of subscripts conflicts with declaration
43	subscript out of range
44	forces location conflict for items in common
45	forces location in negative direction
46	forces location conflict
47	statement number expected
48	CHARACTER and numeric item in same common block
49	CHARACTER and non character item conflict
50	invalid symbol in expression
51	SUBROUTINE name in expression
52	INTEGER or REAL expected
53	INTEGER,REAL or CHARACTER expected
54	types not compatible
55	LOGICAL expression expected
56	too many subscripts
57	too FEW subscripts
58	variable expected
59	"=" expected
60	size of CHARACTER items must agree
61	assignment types do not match
62	SUBROUTINE name expected
63	dummy parameter not allowed
65	assumed size declarations only for dummy arrays
66	adjustable size array declarations only for dummy arrays
67	assumed size must be last dimension
68	adjustable bound must be parameter or in COMMON
69	adjustable bound must be simple integer variable
70	more than one main program
71	size of named COMMON must agree
72	dummy arguments not allowed
73	COMMON variables not allowed
74	SUBROUTINE, FUNCTION or INTRINSIC names not allowed
75	subscript out of range
76	repeat count must be >=1
77	constant expected
78	type conflict
79	number of variables does not match
80	label not allowed
81	no such INTRINSIC FUNCTION

Error Code	Message
82	INTRINSIC FUNCTION type conflict
83	letter expected
84	FUNCTION type conflict with previous call
85	SUBROUTINE / FUNCTION already defined
87	argument type conflict
88	SUBROUTINE / FUNCTION conflict with previous use
89	unrecognizable statement
90	CHARACTER FUNCTION not allowed
91	missing END statement
93	fewer actual than dummy arguments in call
94	more actual than dummy arguments in call
95	argument type conflict
96	SUBROUTINE / FUNCTION not defined
98	CHARACTER size invalid
100	statement order
101	unrecognizable statement
102	jump into block not allowed
103	label already used for FORMAT
104	label already defined
105	jump to FORMAT not allowed
106	DO statement not allowed here
107	DO label must follow DO statement
108	ENDIF not allowed here
109	matching IF missing
110	improperly nested DO block in IF block
111	ELSEIF not allowed here
112	matching IF missing
113	improperly nested DO or ELSE block
114	"(" expected
115	")" expected
116	THEN expected
117	logical expression expected
118	ELSE not allowed here
119	matching IF missing
120	GOTO not allowed here
122	block IF not allowed here
123	logical IF not allowed here
124	arithmetic IF not allowed here
125	"," expected
126	expression of wrong type
127	RETURN not allowed here
128	STOP not allowed here
129	END not allowed here
131	label not defined
132	DO or IF block not terminated
133	FORMAT not allowed here
134	FORMAT label already referenced
135	FORMAT label missing
136	identifier expected
137	integer variable expected
138	TO expected

Error Code	Message
139	integer expression expected
140	ASSIGN statement missing
141	unrecognizable character constant
142	character constant expected
143	integer expression expected
144	STATUS option expected
145	character expression not allowed
146	FILE= missing
147	RECL= already defined
148	integer expression expected
149	unrecognizable option
150	RECL= missing
151	adjustable arrays not allowed here
152	end of statement encountered in implied DO
153	variable required as control for implied DO
154	expressions not allowed in I/O list
155	REC= option already defined
156	integer expression expected
157	END= not allowed here
158	END= already defined
159	unrecognizable I/O unit
160	unrecognizable format in I/O
161	options expected after ","
162	unrecognizable I/O list element
163	FORMAT not found
164	ASSIGN missing
165	label used as FORMAT
166	integer variable expected
167	label defined more than once as format
188	Statement too complicated
203	CHARACTER FUNCTION not allowed
406	unit zero must be formatted and sequential
407	ERR= already defined
408	too many labels
409	invalid size for this type
410	PRECISION expected
411	integer type conflict
415	dimension too big
420	invalid FUNCTION call
421	invalid INTRINSIC FUNCTION
501	unrecognizable character
502	blank not allowed in metacommand
503	metacommand not allowed here
504	size already defined
601	out of range
701	CHARACTER type expected
703	internal error
705	internal error
706	internal error
708	internal error
709	CHARACTER type not expected

Error Code	Message
710	internal error
711	internal error
713	long integer conversion error
718	internal error
802	invalid radix
811	numeric or character expected
812	COMMON exceeds max size
813	array dimension upper bound < lower bound

A.3 RUNTIME ERROR MESSAGES

Runtime errors are divided into two classes:

- file system errors
- nonfile system errors

Nonfile system errors include the following:

- memory errors
- type REAl arithmetic errors
- type INTEGER*4 arithmetic errors
- other errors

If you find a runtime error code which is not listed in this table, it may be an operating system error. Refer to the Wang PC Introductory Guide, 700-8020, for information on operating system errors.

A.3.1 File System Errors

Code numbers 1000 through 1099 are status code, always issued in conjunction with an OS status code.

Error Code	Message
1000	write error when writing end of file
1002	filename extension with more than 3 characters
1003	error during creating of new file (Disk or directory full)
1004	error during open of existing file (File not found)
1005	filename with more than 21 character or contains invalid characters
1007	total filename length over 21 characters
1008	write error when advancing to next record
1009	file too big (Over 65535 logical sectors)
1010	write error when seeking to direct record device
1011	attempt to open a random file to a non-disk device
1012	forward space or back space on a non-disk device
1013	disk or director full error during forward space or back space

Error Code	Message
1200	format missing final ")"
1201	sign not expected in input
1202	sign not followed by digit in input
1203	digit expected in input
1204	missing N or Z after B in format
1205	unexpected character in format
1206	zero repetition factor in format not allowed
1207	integer expected for w field in format
1208	positive integer required for w field in format
1209	"." expected in format
1210	integer expected for d field in format
1211	integer expected for e field in format
1212	positive integer required for e field in format
1213	positive integer required for w field in a format
1214	hollerith field in format must not appear for reading
1215	hollerith field in format requires repetition factor
1216	X field in format requires repetition factor
1217	P field in format requires repetition factor
1218	integer appears before + or - in format
1219	integer expected after + or - in format
1220	P format expected after signed repetition factor in format
1221	maximum nesting level for formats exceeded
1222	")" has repetition factor in format
1223	integer followed by , illegal in format
1224	"." is illegal format control character
1225	character constant must not appear in format for reading
1226	character constant in format must not be repeated
1227	"/" in format must not be repeated
1228	"\" in format must not be repeated
1229	BN or BZ format control must not be repeated
1230	attempt to perform I/O on unknown unit number
1231	formatted I/O attempted on file opened as unformatted
1232	format fails to begin with "("
1233	I format expected for integer read
1234	F or E format expected for real read
1235	two "." characters in formatted real read
1236	invalid REAL number
1237	L format expected for logical read
1238	blank logical field
1239	T or F expected in logical read
1240	A format expected for character read
1241	I format expected for integer write
1242	W field in F format not greater than d field + 1
1243	scale factor out of range of d field in E format
1244	E or F format expected for real write
1245	L format expected for logical write
1246	A format expected for character write
1247	attempt to do unformatted I/O to a unit opened as formatted
1251	integer overflow on input
1252	too many bytes read from input record
1253	too many bytes written to direct access unit record
1255	attempt to do external I/O on a unit beyond end of file record

Error Code	Message
1256	attempt to position a unit for direct access on a nonpositive record number
1257	attempt to do direct access to a unit opened as sequential
1258	unable to seek to file position
1260	attempt to BACKSPACE, REWIND or ENDFILE unit connected to unblocked device
1261	premature end of file of unformatted sequential file
1262	invalid blocking in unformatted sequential file
1263	?? format not permitted on internal unit
1264	attempt to do unformatted I/O to internal unit
1265	attempt to put more than one record into internal unit
1266	attempt to write more characters to internal unit than its length
1267	EOF called on unknown unit
1268	dynamic file allocation limit exceeded
1269	scratch file opened for read
1270	console I/O error
1271	EOF function called on terminal device
1272	file operation attempted after error encountered on previous operation
1273	keyboard buffer overflow: too many bytes written to keyboard input record (must be less than 132)
1274	reading long integer
1275	writing long integer
1281	repeat field not on integer
1282	multiple repeat character
1284	list directed numeric items bigger than record size
1298	end of file encountered
1299	Integer variable not ASSIGNed a label used in assigned GOTO

A.3.2 Other Runtime Errors

Nonfile system error codes range from 2000 to 2999. In some cases, metacommands determine if errors are checked; in other cases, error codes are always checked.

A.3.3 Memory Errors

The heap is the storage area that PC FORTRAN dynamically allocates storage for file control blocks. Since the stack and the heap grow towards each other, these errors are all related; for example, a stack overflow can cause a "Heap is Invalid" error.

Code	Message
2000	stack overflow
2002	heap is invalid
2052	signed divide by zero (This error appears only when $DEBUG is set.)
2054	signed math overflow
2084	INTEGER zero to negative power

Type REAL Arithmetic Errors

2100	REAL divide by zero
2101	REAL math overflow
2102	SIN or COS argument range
103	EXP argument range
2104	SQRT of negative argument
2105	LN of non-positive argument
2106	TRUNC/ROUND argument range
2131	tangent argument too small
2132	arcsin or arccosine of REAL > 1.0
2133	negative REAL to REAL power
2134	REAL zero to negative power
2135	REAL math underflow
2136	REAL indefinite (uninitialized or previous error)
2137	Missing arithmetic processor (You have linked your program with the runtime library intended for use with the 8087 numeric coprocessor, but there is no coprocessor on your system. Relink your program with the runtime library that emulates floating point arithmetic.)
2138	REAL IEEE denormal detected (A very small real number was generated and may no longer be valid due to loss of significance.)
2139	REAL precision loss (An arithmetic operation on the 8087 numeric coprocessor has generated a loss of numeric precision in the result of an operation.)
2140	REAL arithmetic processor instruction illegal or not emulated (An attempt was made to execute an illegal arithmetic coprocessor instruction, or the floating point emulator cannot emulate a legal coprocessor instruction.)

Type Integer*4 Arithmetic Errors

2200	long integer divided by zero
2201	long integer math overflow
2234	long integer zero to negative power

A.3.4 Other Errors

2451	Assigned GOTO label not in list (This error appears only when $DEBUG is set.)

APPENDIX B
PC FORTRAN AND ANSI SUBSET FORTRAN 77

B.1 INTRODUCTION

This appendix describes how Wang Professional Computer FORTRAN differs from the standard subset language. There are two levels in the standard: full FORTRAN and subset FORTRAN. Wang Professional Computer FORTRAN is a subset of the latter. The differences between Wang Professional Computer FORTRAN and the standard subset fall into two general categories: full language features and extensions to the standard.

B.2 FULL LANGUAGE FEATURES

Several features from the full language are included in Wang PC FORTRAN. A program written to comply with the subset restrictions compiles and executes properly in Wang PC FORTRAN. The differences between Wang PC FORTRAN and the subset standard are discussed in this section.

Subscript Expressions

While the subset does not allow function calls or array element references in subscript expressions, they are allowed in the full language and in Wang PC FORTRAN.

DO Variable Expressions

The subset does not any allow expressions rather than simple integers in a DO statement; the full language and Wang PC FORTRAN do. Wang PC FORTRAN further allows integer expressions in implied DO loops associated with READ and WRITE statements.

Unit I/O Number

Both Wang PC FORTRAN and the full language allow you to specify an I/O unit in an integer expression.

Expressions in Input/Output List

The subset does not allow expressions to appear in an I/O list while the full language allows expressions in the I/O list of WRITE statements. Wang PC FORTRAN allows expressions in the I/O list of a WRITE statement providing that the expressions do not begin with an initial left parentheses. Expressions like (A+B)*(C+D) can be specified in an output list as +(A+B)*(C+D) without generating any extra code.

Double Precision

PC FORTRAN provides for double precision real numbers as in the full language.

Edit Descriptors

PC FORTRAN allows for D and G edit descriptors as in the full language.

Expressions in Computed GOTO

PC FORTRAN and the full language allow an expression for the selector of a computed GOTO.

Generalized I/O

Wang PC FORTRAN allows formatted or unformatted files. In the subset, direct access files must be unformatted while sequential files must be formatted.

Wang PC FORTRAN contains an augmented OPEN statement that takes parameters not included in the subset. In addition, one form of the CLOSE statement is not included in the subset. The READ and WRITE statements allow the optional ERR parameter.

B.3 EXTENSIONS TO STANDARD

The implemented language has several minor extensions to the full language standard. These are as follows:

User-defined Names

User-defined names greater than six characters are allowed, although only the first six characters are significant.

Tabs in Source Files

Tabs are allowed in the source file. For more information, refer to Subsection 2.6.5, Tabs.

Compiler Metacommands

Metacommands or compiler directives allow you to communicate certain information to the compiler. A metacommand line conveys information about the nature of the current compilation to the FORTRAN system. The metacommand begins with a dollar sign $ in column 1 and can generally appear any place that a comment line can appear although certain metacommands are restricted as to their location.

NOTSTRICT

The standard is relaxed when the $NOTSTRICT metacommand is in effect. This relaxation allows PC FORTRAN features such as assignment of character to any variable type and initialization of any variable with character data.

Backslash Edit Control

The backslash (\) edit control character inhibits normal advancement to the next record associated with the completion of a READ or WRITE statement. The backslash is useful when prompting to an interactive device such as the screen so that a response can appear on the same line as the prompt.

End Of File Intrinsic Function

The EOF function accepts a unit specifier as an argument and returns a logical value that indicates whether the specified unit is at its end of file.

Lowercase Input

The processor treats lowercase characters exactly like uppercase characters except in character constants and Hollerith fields.

Binary Files

Binary files are similar to unformatted sequential files except that they have no internal structure. This allows the program to create or read files with arbitrary contents. which is helpful for files created by or intended for programs written in languages other than FORTRAN.

APPENDIX C
PC FORTRAN FILE CONTROL BLOCK

This appendix lists the complete file control block specification for this version of the PC FORTRAN runtime system. The underlying data type is a PC Pascal record. Numbers in square brackets give the byte offset for each field of the file control block.

PC FORTRAN File Control Block for MS-DOS

```
INTERFACE;  UNIT
  FILKQQ (FCBFQQ, FILEMODES, SEQUENTIAL, TERMINAL,
          DIRECT, DEVICETYPE, CONSOLE, LDEVICE,
          DISK, DOSEXT, DOSFCB, FNLUQQ, SCTRLNTH);

CONST

  FNLUQQ = 21;          {length of an MS-DOS filename}
  SCTRLNTH = 512;       {length of a disk sector}
TYPE

DOSEXT = RECORD         {DOS file control blck extension}

  PS[0]: BYTE;          {boundary byte, not in extension}
  FG[1]: BYTE;          {flag;  must be 255 in extension}
  XZ[2]: ARRAY [0..4] OF BYTE       {pad, internal use}
  AB[7]: BYTE;          {internal use for attribute bits}
END;

DOSFCB = RECORD         {DOS file control block (normal)}

  DR[0]: BYTE;          {drive numb, 0=default, 1=A etc}
  FN[1]: STRING (8);    {file name - eight characters}
  FT[9]: STRING (3);    {file extn - three characters }
  EX[12]: BYTE;         {current extent;  lo-order byte}
  E2[13]: BYTE;         {current extent;  hi-order byte}
  S2[14]: BYTE;         {size of sector;  lo-order byte}
  RC[15]: BYTE;         {size of sector;  hi-order byte}
  Z1[16]: WORD;         {file size; lo-word; readonly}
  Z2[18]: WORD;         {file size; hi-word; readonly}
  DA[20]: WORD;         {date; bits: DDDDDMMMMYYYYYYY}
  DN [22]: ARRAY [0..9] OF BYTE;  ↔reserved for DOS}
  CR[32]: BYTE;         {current sector (within extent)}
  RN[33]: WORD;         {direct sector number (lo word)}
```

```
  R2[35]: BYTE;        {direct sector number (hi byte}
  R3[36]: BYTE;        {DSN hi byte if sect size < 64}
  PD[37]: BYTE;        {pad to word boundary; not MS-DOS}
END;

DEVICETYPE=(CONSOLE, LDEVICE, DISK); {physical device}

FILEMODES=(SEQUENTIAL, TERMINAL, DIRECT); {access mode}

TYPE

FCBFQQ=RECORD  {byte offsets start every field comment}

{fields accessible as <file variable>.<field>}
TRAP: BOOLEAN;     {00 Pascal user trap errs if true}
ERRS: WRD(0)..15; {01 err stat, set only by all units}
MODE: FILEMODES;  {02 usr file mode; not used in unitU}
MISC: BYTE;       {03 pad to word bound, special use}

{flds shared by units F, V, U; ERRC/ESTS are write-only}
ERRC: WORD;       {04 err code, err exists if nonzero}
                  {1000..1099:  set for unit U errors}
                  {1100..1199:  set for unit F errors}
                  {1200..1299:  set for unit V errors}
ESTS: WORD;       {06 error data, usually set by OS}
CMOD: FILEMODES;  {08 sys file mode; copied from MODE}

{flds set/used by units F and V, read-only in unitU}
TXTF: BOOLEAN; {09 true: formatted /ASCII /TEXT file}
               {   false: not formatted / binary file}
SIZE: WORD;    {10 record size set when file is opened}
               {DIRECT: always fixed record length}
               {others: max length (UPPER (BUFFA))}
MISB: WORD;    {12 unused, exists for historic reasons}
OLDF: BOOLEAN; {14 true: must exist before open; RESET}
               {   false: can create on open; REWRITE}
INPUT: BOOLEAN; {15 true: user is now reading from file}
               {   false: user is now writing to file}
RECL: WORD;    {16 DIRECT record number, lo order word}
RECH: WORD;    {18 DIRECT record number, hi order word}
USED: WORD;    {20 number bytes used in current record}

{fld usd internally by units F and V not used by unitU}
LINK: ADR OF FCBFQQ;{22 DS offset addr of next opn fil}
{flds usd internally by unitF not used by units V or U}
BADR: ADRMEM;  {24 ADR of buffer variable (end of FCB)}
TMPF: BOOLEAN; {26 true if temp file;  delete on CLOSE}
FULL: BOOLEAN; {27 buffer lazy evaluation status, TEXT}
UNFM: BYTE;    {28 for unformatted binary mode}
OPEN: BOOLEAN; {29 file opened; RESET / REWRITE called}

{flds usd internally by unitV not used by units F or U}
FUNT: INTEGER; {30 Unit V's unit number always above 0}
ENDF: BOOLEAN; {32 last operation was the ENDFILE stmt}
```

```
{flds set/usd by unitU, and read-only in units F and V}
REDY: BOOLEAN; {33 buffer ready if true;  set by F / U}
BCNT: WORD;    {34 number of data bytes actually moved}
EORF: BOOLEAN; {36 true if end of record read, written}
EOFF: BOOLEAN; {37 end of file flag set after EOF read}

    {unit U (operating system) information starts here}

NAME: LSTRING(FNLUQQ);{38  DOS filename(D:NNNNNNNN.XXX)}
DEVT: DEVICETYPE;     {60  device type, accessed by file}
RDFC: BYTE;           {61  funct code, for device GET}
WRFC: BYTE;           {62  funct code, for device PUT}
CHNG: BOOLEAN;        {63  true if sbuf data was chgd}
SPTR: WORD;           {64  index to current byte in sbuf}
LNSB: WORD;           {66  num of valid bytes in sbuf}
DOSX: DOSEXT;         {68  extend DOS file contrl blck}
DOSF: DOSFCB;         {76  normal DOS file contrl blck}
IEOF: BOOLEAN;        {114 true if eoff will be true}
                           {on next get}
FNER: BOOLEAN;        {115 true if pfnuqq filename err}
SBFL: BYTE;           {116 max txtfil line len in sbuf}
SBFC: BYTE;           {117 num of chars, read to sbuf}
SBUF: ARRAY[WRD(0)..SCTRLNTH-1] OF BYTE; {118 sect buf}
PMET: ARRAY [0..3] OF BYTE; {118+sctrlnth reserved pad}
BUFF: CHAR;                 {122+sctrlnth (buffer var)}

    {end of section for unit U specific OS information}

END;
END;
```

APPENDIX D
REAL NUMBER CONVERSION UTILITIES

Releases of PC FORTRAN starting with version 3.0 and later use the IEEE real number format. Releases of PC FORTRAN earlier than 3.0 used the real number format. The two formats are not compatible. However, if you need to convert real numbers from one format to the other, you can use the following library routines:

1. Wang to IEEE format

   SUBROUTINE M2ISQQ (RMS , RIEEE)

2. IEEE to Wang format

   SUBROUTINE I2MSQQ (RIEEE , RMS)

RMS and RIEEE are real numbers in Wang format and in IEEE format, respectively.

APPENDIX E
STRUCTURE OF EXTERNAL PC FORTRAN FILES

The structure of an external PC FORTRAN file is determined by its properties. The structures used in PC FORTRAN are as follows.

- Formatted sequential files

  Records are separated by carriage return and linefeed (ASCII hex codes 0D and 0A, respectively).

  record N D A record N+1

- Unformatted sequential files

  A logical record is represented as a series of physical records, each of which has the following structure.

  L data <= 128 bytes L

  physical record

  Each L shown above is a length byte that indicates the length of the data portion of the physical record. The data portion of the last physical record contains MOD (length of logical record, 128) bytes, and the length bytes will contain the exact size of the data portion.

  Each of the preceding physical records will contain 128 bytes in the data portion, while the length byte will contain 129. For example, if the size of the logical record is 138:

  129 128 bytes of data 129 10 10 bytes of data 10

  <---------- 1 logical record ----------->

  The first byte of the file is reserved and contains the value 75, which has no other significance.

- Formatted direct files, unformatted direct files, and binary files

  No record boundaries or any other special characters are used.

APPENDIX F
PC FORTRAN SCRATCH FILE NAMES

Scratch files are created by the PC FORTRAN system when no filename is specified in an OPEN statement. Scratch file names look like this:

```
T<u>.TMP
```

<u> is the unit number specified in the OPEN statement.

APPENDIX G
CUSTOMIZING i8087 INTERRUPTS

This appendix describes how to customize the i8087 interrupts on your computer system. Before proceeding, you should be familiar with the following:

- The Intel publication, iAPX 86/20, 88/20 Numeric Supplement
- The PC Assembler
- DEBUG, the PC debugger utility

In addition, make backup copies of any of the disks you plan to modify.

To change the way the runtime library processes interrupts, you must use the PC debugger DEBUG (or a similar utility). Although this utility is intended primarily for debugging assembly language programs, you can also use it to alter the binary contents of any file. You will use this second capability of DEBUG to to customize FORTRAN.L87 for a particular hardware configuration.

FORTRAN.L87, the 8087 version of the runtime library, contains the following assembly language structure:

```
i8087control STRUC
LABX87      DB   '<8087>';48-bit tag
EOIX87      DB   0       ;EOI instruction
PRTX87      DB   0       ;i8259 port number
SHRX87      DB   0       ;Shared interrupt device
INTX87      DB   2       ;i8087 interrupt vector #
INTOFFSET   DW   0       ;
i8087control ENDS
```

This structure defines the default control values used by the runtime library to handle 8087 interrupts. Each of the elements of the structure is described briefly.

- LABX87

  A string label. LABX87 exists solely to locate the other structure fields in the executable binaries and libraries.

- EOIX87

  The hexadecimal value of the i8259 "end of interrupt" instruction for a particular implementation. To the 8087 interrupt handler supplied by Wang, any nonzero value of this byte indicates the presence of an i8259 interrupt controller.

- PRTX87

  The control port number associated with an i8259, if present.

- SHRX87

  If nonzero, an indication that the i8087 shares its interrupt vector with another device. In such a case, when the 8087 interrupt handler supplied by Wang determines that an interrupt it receives is not an 8087 interrupt, it passes control to the other interrupt device.

- INTX87

  The interrupt vector number to which the 8087 is connected.

Depending on the setup of your computer system, any or all (or none) of the last four items may require changing. Specifically, you must alter this structure if your hardware configuration meets any of the following criteria:

- It uses an 8087 interrupt vector number other than 2.
- It uses an 8259 interrupt controller.
- The 8087 shares interrupts with another device on the same vector.

The example on the following pages demonstrates how to change all of the interrupt parameters on the 8087. In the example, the following specific changes are made:

- The 8087 interrupt control block is altered to set EOIX87 to 255 decimal, thus informing the software that an i8259 exists and that its EOI instruction is 255.
- The i8259 should issue its EOI request through port number 254 (PRTX87).
- The nonzero value of SHRX87 indicates that the 8087 shares its interrupts with another device.
- The interrupt vector number of the i8087 was changed to 4.

These values are used merely for the purpose of this sample session. Consult your hardware manual for the values required for your computer system.

Sample DEBUG Session to Customize i8087 Interrupts

```
1.  >

2.  >debug b:fortran.187

3.  DEBUG-86  version 2.10

4.  >r
    AX=0000  BX=0001  CX=B800  DX=0000
    SP=FFEE  BP=0000  SI=0000  DI=0000
    DS=0AF9  ES=0AF9  SS=0AF9  CS=0AF9
    IP=0100   NV UP DI PL NZ NA PO NC
    0AF9:0100 F0            LOCK
    0AF9:0101 FD            STD

5.  >s ds:100 lefff '8087>'

6.  0AF9:2370

7.  >d af9:2370
    0AF9:2370  38 30 38 37 3E 00 00 00   8087>...
               02 00 00 F8 A0 DF 00 02   ...x ►..

8.  >d af9:2375
    0AF9:2375  00 00 00-02 00 00 F8 A0   .....x
              DF 00 02                   ►..

9.  >e af9:2375
    0AF9:2375 00.ff  00.fe  00.1
    0AF9:2378 02.4

10. >d af9:2375
    0AF9:2375  FF FE 01-04 00 00 F8 A0   ......x
               DF 00 02                  ►..

11. >w

12. >q

13. >
```

Only the parts of the screen display that apply specifically to this procedure are shown above. Numbers 1 through 13 on the left margin do not appear on the screen. They refer to the corresponding numbers comments on the page following the sample session.

Comments for Sample DEBUG Session

1. PC prompt.

2. Call DEBUG with FORTRAN.L87.

3. DEBUG utility prompt.

4. Instruct debugger to show 8086 registers.

5. Instruct debugger to search efff bytes beginning at DS:100 for the string '8087>'.

6. String found at 0AF9:2370.

7. Instruct debugger to display the string.

8. Advance to the beginning of the 'i8087control' structure.

9. Instruct the debugger to make the following alterations:

```
EOIX87 to FF hex, 255 decimal
PRTX87 to FE hex, 254 decimal
SHRX87 to 1 hex
INTX87 to 4
```

10. Instruct the debugger to display any changes.

11. Write any changes to the source file.

12. Stop the debugger.

13. DOS prompt returns.

APPENDIX H
PC LINK ERROR MESSAGES

Any link error will cause the link session to abort. After you have found and corrected the problem, you must rerun MS-LINK. Link errors have no code number. Refer to your your Wang PC Program Development Guide, 700-8018 for further information on the Linker.

Attempt to access data outside of segment bounds, possibly bad object module

There is probably a bad object file.

Bad numeric parameter

Numeric value is not in digits.

Cannot open temporary file

The Linker is unable to create the file VM.TMP because the disk directory is full. Insert a new disk. Do not remove the disk that will receive the list map file.

Error: dup record too complex

DUP record in assembly language module is too complex. Simplify DUP record in assembly language program.

Error: fixup offset exceeds field width

An assembly language instruction refers to an address with a short instruction instead of a long instruction. Edit assembly language source and reassemble.

Input file read error

There is probably a bad object file.

Invalid object module

An object module(s) is incorrectly formed or incomplete (as when assembly is stopped in the middle).

Symbol defined more than once

The Linker found two or more modules that define a single symbol name.

Program size or number of segments exceeds capacity of linker

The total size may not exceed 384K bytes and the number of segments may not exceed 255.

Requested stack size exceeds 64K

Specify a size greater than or equal to 64K bytes with the /STACK switch.

Segment size exceeds 64K

64K bytes is the addressing system limit.

Symbol table capacity exceeded

Very many and/or very long names were typed, exceeding the limit of approximately 25K bytes.

Too many external symbols in one module

The limit is 256 external symbols per module.

Too many groups

The limit is 10 groups.

Too many libraries specified

The limit is 8 libraries.

Too many public symbols

The limit is 1024 public symbols.

Too many segments or classes

The limit is 256 (segments and classes taken together).

Unresolved externals: <list>

The external symbols listed have no defining module among the modules or library files specified.

VM read error

This is a disk error; it is not caused by the Linker

Warning: no stack segment

None of the object modules specified contains a statement allocating stack space, but you used the /STACK switch.

Warning: segment of absolute or unknown type

There is a bad object module or an attempt has been made to link modules that MS-LINK cannot handle (e.g., an absolute object module).

Write error in TMP file

No more disk space remains to expand VM.TMP file.

Write error on run file

Usually, there is not enough disk space for the run file.

APPENDIX I
INTERFACING TO ASSEMBLY LANGUAGE ROUTINES

All subroutines and functions in PC FORTRAN are external. They need not be declared as external with the EXTERNAL statement. When a subroutine or function is called, the addresses of the actual parameters are first pushed on the stack in the order that they are declared. PC FORTRAN always uses calls by reference, even if the actual parameters are expressions or constants.

If the procedure called is a function and if the function return type is real or double precision, an additional implicit parameter for the function is pushed on the stack. This parameter is the two-byte address of a temporary variable created by the calling program.

After all parameters have been pushed, the return address is pushed. If the procedure called is a function, the return value is expected as follows:

- If the return value is a two-byte integer or logical value, that value is expected in the AX register, as shown in Figure G-1:

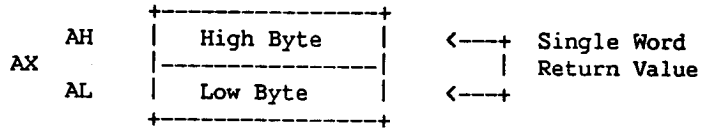

Figure G-1 Two-Byte Return Value

- If the return value is a four-byte integer or logical value, that value is expected in the DX, AX pair, as shown in Figure G-2.

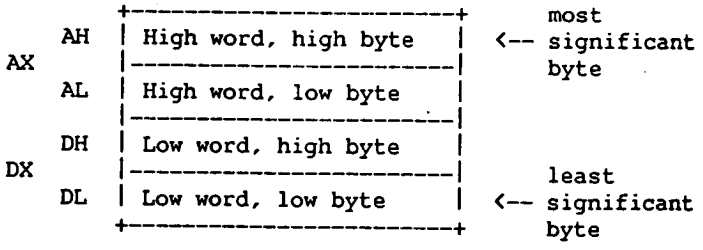

Figure G-2 Four-Byte Return Value

- If the return value is a four-byte or eight-byte REAL value, that value is expected in the temporary variable created by the calling program. The two-byte address of this temporary variable is the last parameter pushed on the stack. It is always at BP+6 (see Example 2).

Example 1. INTEGER*4 Add Routine

Assume the following PC FORTRAN program has been compiled:

```
PROGRAM  EXAMPLE1
INTEGER I, TOTAL, IADD
I = 10
TOTAL = IADD (I,15)
WRITE (*,'(1X,I6)') TOTAL
END
```

At runtime, just prior to the transfer to IADD, the stack would be as shown in Figure G-3.

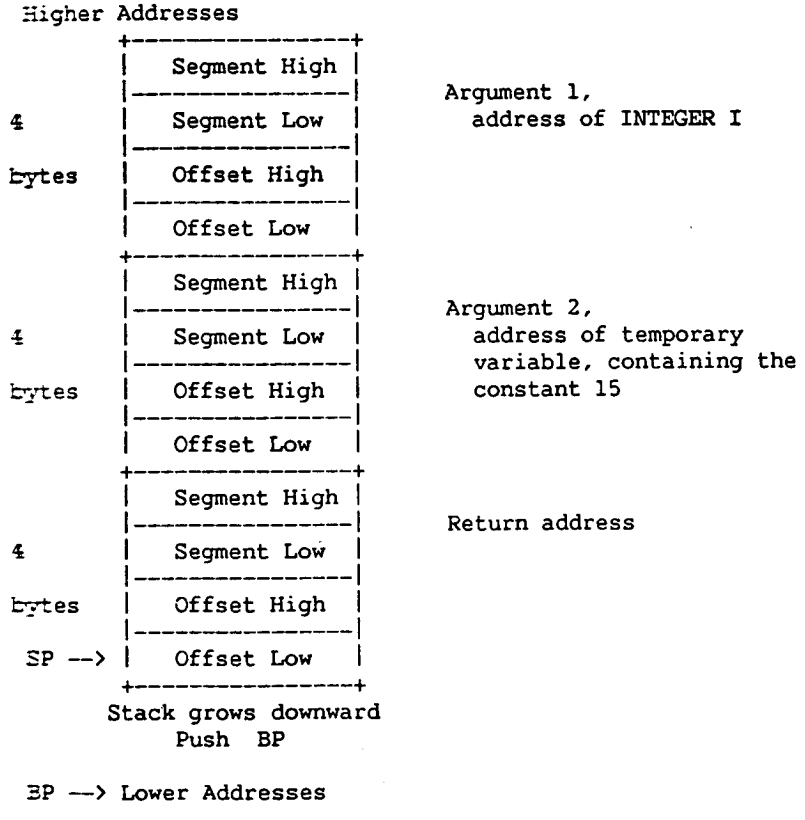

Figure G-3. Stack Before Transfer to IADD

An example of an assembly language routine that implements the integer ADD function, IADD, is illustrated in the following routine. Note that the function return value is AX,DX.

```
DATA    SEGMENT PUBLIC 'DATA'
        ;See Note at end of this section.
DATA    ENDS

DGROUP  GROUP PUBLIC DATA  ;See Note.
CODE    SEGMENT 'CODE'
        ASSUME  CS:CODE,DS:DGROUP,SS:DGROUP ;See Note.
PUBLIC  IADD
IADD    PROC FAR

        PUSH    BP          ;Save framepointer on stack
        MOV     BP,SP
        LES     BX,[BP+10] ;ES,BX := addr of 1st param
        MOV     AX,ES:[BX] ;AX,DX := addr of 1st param
        MOV     DX,ES:[BX]+2
        LES     BX,[BP+6]  ;ES,BX := addr of 2nd param
        ADD     AX,ES:[BX] ;AX,DX := 1st parameter plus
        ADC     DX,ES:[BX]+2  ;2nd parameter
        MOV     SP,BP
        POP     BP          ;Restore the framepointer
        RET     AH          ;Return, pop 8 bytes

        IADD    ENDP
        CODE    ENDS

END
```

Example 2. REAL*4 Add Routine

Assume the following FORTRAN program has been compiled:

```
PROGRAM  EXAMPLE2
REAL R, TOTAL, RADD
R = 10.0
TOTAL = RADD (15.0,R)
WRITE (*,'1X,F10.3)') TOTAL
END
```

At runtime, just prior to the transfer to RADD, the stack would be as shown in Figure G-4:

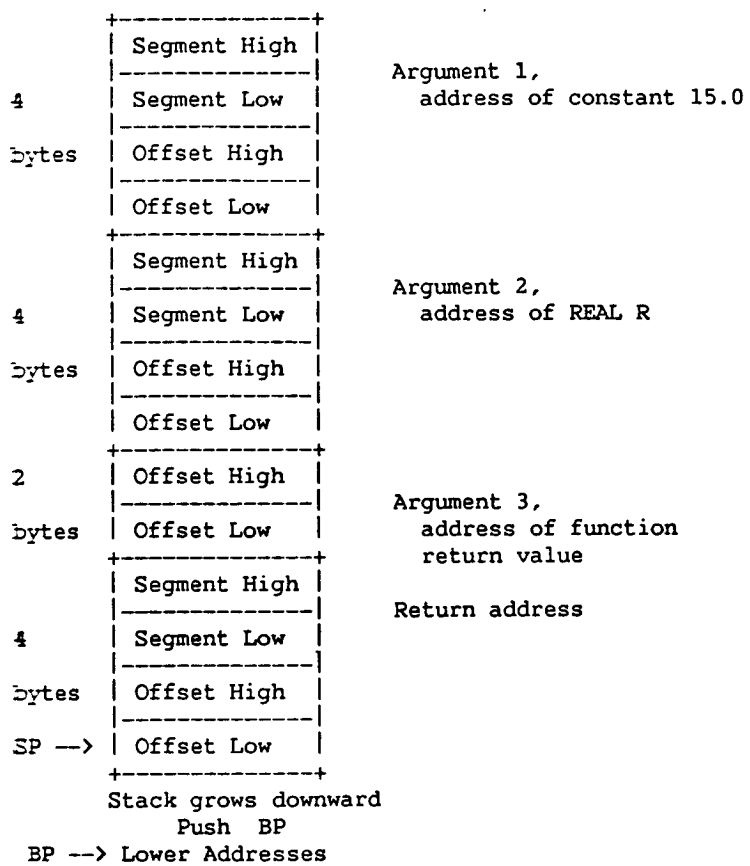

Figure G-4. Stack Before Transfer to RADD

An example of an assembly language routine that implements the real add function, RADD, is illustrated in the following routine. Note that the function return value is in the location specified by BP+6.

```
DATA    SEGMENT PUBLIC 'DATA'
        ;See Note at end of section
DATA    ENDS

DGROUP  GROUP   PUBLIC DATA ;See Note
CODE    SEGMENT 'CODE'
        ASSUME  CS:CODE,DS:DGROUP,SS:DGROUP ;See Note
PUBLIC  RADD
RADD    PROC    FAR
        PUSH    BP          ;Save framepointer on stack
        MOV     BP,SP
        LES     BX,[BP+12] ;ES,BX := addr of 1st param
        FLD     ES:[BX]     ;Push value of 1st param
                            ;on 8087 stack
        LES     BX,[BP+8]  ;ES,BX := addr of 2nd param
        FLD     ES:[BX]     ;Push value of 2nd param
                            ;on 8087 stack
        FADDP   ST(1),ST    ;Add first two items
                            ;on 8087 stack
        MOV     DI,[BP+6]  ;DI := addr of funct return
        FSTP    [DI]        ;Store result on 8087 stack
                            ;at funct return location
        FWAIT
        MOV     SP,BP       ;Restore the framepointer
        POP     BP
        RET     0AH         ;Return, pop 10 bytes

        RADD    ENDP
        CODE    ENDS

END
```

APPENDIX J
ASCII CHARACTER CODES

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hec	CHR
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	,	087	57H	W
002	02H	STX	045	2DH	-	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	3OH	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	\
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	^
009	09H	HT	052	34H	4	095	5FH	
010	0AH	LF	053	35H	5	096	60H	`
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	"	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	{

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hec	CHR
038	26H	&	081	51H	Q	124	7CH	|
039	27H	'	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	~
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

Dec=Decimal, Hex=Hexadecimal (H), CHR=Character.
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

INDEX

D

E

F

G

H

I

L

M

N

O

P

R

S

T